Visual Component Library Reference Volume 1

Borland®

Delphi™ 3

for Windows 95 & Windows NT

Visual Component Library Reference

VOLUME 1

Borland®
Delphi™
for Windows 95 and Windows NT

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1983, 1997 Borland International. All rights reserved. All Borland product names are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

HDA1330WW21773 1E0R0297
9798990001-9 8 7 6 5 4 3 2 1
H1

Contents

Chapter 1
Visual Component Library Reference 1

Visual Component Library 1
Manual conventions 2
Using these manuals. 2
Software registration and technical support . . . 2
TActiveForm 3
TAnimate 8
TApplication 21
TAutoIncField 46
TBatchMove 50
TBCDField 61
TBDEDataSet 70
TBevel 91
TBinaryField 95
TBitBtn 100
TBlobField 109
TBooleanField 119
TButton 125
TButtonControl. 131
TBytesField 134
TCGIApplication. 138
TCheckBox 141
TCheckListBox 145
TClientDataSet 150
TClientSocket. 181
TColorDialog 185
TComboBox 189
TCommonDialog. 193
TComponent 198
TControl. 217
TCoolBar 263
TCurrencyField. 271
TCustomCheckBox 275
TCustomComboBox 281
TCustomContentProducer 295
TCustomControl 298
TCustomDataStore. 303
TCustomDBGrid 317
TCustomDecisionGraph 348
TCustomDecisionGrid 357
TCustomEdit 376
TCustomForm 390
TCustomGrid. 423
TCustomGroupBox 452
TCustomHotKey 456
TCustomImageList 462
TCustomLabel. 481
TCustomListBox 488
TCustomListView. 507
TCustomMaskEdit 534
TCustomMemo 548
TCustomPageProducer. 557
TCustomPanel. 562
TCustomProvider. 572
TCustomRadioGroup. 578
TCustomRemoteServer. 583
TCustomRichEdit 590
TCustomServerSocket 606
TCustomSocket 619
TCustomStaticText 632
TCustomTabControl 637
TCustomTreeView 646
TCustomUpDown 663
TCustomWebDispatcher 673
TDatabase 680
TDataModule 698
TDataSet 702
TDataSetTableProducer 744
TDataSetUpdateObject 747
TDataSource 749
TDateField 755
TDateTimeField 760
TDateTimePicker 768
TDBCheckBox 777
TDBComboBox 784
TDBCtrlGrid. 791
TDBCtrlPanel 804
TDBDataSet 808
TDBEdit 815
TDBGrid 824
TDBImage 829
TDBListBox 838
TDBLookupComboBox. 845
TDBLookupControl. 852
TDBLookupListBox. 859

Index Volume 2

Chapter

1

Visual Component Library Reference

These manuals, volumes I and II, are the reference for the Delphi *Visual Component Library* (VCL). They include all components that can be placed in a form or a data module and their base classes *TObject* and *TPersistent*. Use them when you want to look up the details of a particular VCL component or its properties, methods, and events.

To look up information about types, variables, constants, global routines, or objects that do not descend from *TComponent*, see Help.

Visual Component Library

The VCL is made up of objects, most of which are also components. Using the objects and components of VCL, you are unlimited in the range of Windows programs you can rapidly develop.

A Delphi object consists of properties, methods, and events. Properties represent the data contained in the object. Methods are the actions the object can perform. Events are conditions the object can react to. All objects descend from the ancestor object *TObject*.

Components are visual objects that you can manipulate at design time. All components descend from the *TComponent* object. To program with a component, this is the model you will use most frequently:

1. Select a component from Delphi's Component palette and add it to a form.
2. Set property values of the component using the Object Inspector.
3. Respond to events that might occur to the component at runtime. To respond to an event, you write code within an event handler. Your code modifies property values and calls methods.

For detailed information on how to perform these three steps, see the Delphi *User's Guide*.

You can create your own objects and components by deriving them from the existing Delphi objects and components. For information about writing your own components, see the Delphi *Developer's Guide*.

Manual conventions

The printed manuals for Delphi use the typefaces and symbols described here to indicate special text.

Table 1.1 Typefaces and symbols in these manuals

Typeface or symbol	Meaning
`Monospace type`	Monospaced text represents text as it appears onscreen or in Object Pascal code. It also represents anything you must type.
Boldface	Boldfaced words in text represent Object Pascal reserved words or compiler directives.
Italics	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
Keycaps	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."

Using these manuals

These manuals are organized by object. Each entry provides a brief object description followed by navigational tables that list all the properties, events, and methods that you can use when working with the object. Individual properties, events, and methods are described under the object that introduces them. For example, the *Height* and *Width* properties, common to all controls, are introduced by *TControl*. Entries describing these properties are under the entry for *TControl*. All other objects that surface these properties point to *TControl* in their navigational tables.

Software registration and technical support

The Borland Assist program offers a range of technical support plans to fit the diverse needs of individuals, consultants, large corporations, and developers. To receive help with this product, return the registration card and select the Borland Assist plan that best suits your needs. North American customers can register by phone 24 hours a day at 1-800-845-0147. For additional details on these and other Borland services, see the *Borland Assist Support and Services Guide* included with this product. To find the location of the support office in your region, visit the Borland Web site at www.borland.com.

TActiveForm

Unit: AXCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TScrollingWinControl • TCustomForm • TActiveForm

A *TActiveForm* object is a Delphi form that is created via a class factory and used as an ActiveX control in ActiveX host applications.

Description In Delphi, a *TActiveForm* object is used to create an form ActiveX control that can be displayed and run from within Web browsers.

See also *TActiveFormControl* object, *TActiveFormFactory* object

Properties

The following table lists the properties of *TActiveForm*:

Property	Visibility	Default	Documented in
Active	public, read-only		TCustomForm
ActiveControl	published		TCustomForm
ActiveOleControl	public		TCustomForm
Align	public	alNone	TControl
AutoScroll	published	True	TScrollingWinControl
AxBorderStyle	published	afbSingle	TActiveForm
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomForm
Caption	public		TControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TCustomForm
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl

Property	Visibility	Default	Documented in
Designer	public		TCustomForm
DesignInfo	public		TComponent
DropTarget	public		TCustomForm
Enabled	public	True	TControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
HorzScrollBar	published		TScrollingWinControl
KeyPreview	published	False	TCustomForm
Left	published		TControl
Menu	public		TCustomForm
ModalResult	public		TCustomForm
Name	published		TControl
OleFormObject	public		TCustomForm
Owner	public, read-only		TComponent
Parent	public		TCustomForm
ParentWindow	public		TWinControl
PixelsPerInch	published		TCustomForm
PopupMenu	published		TControl
PrintScale	published	poProportional	TCustomForm
Scaled	published	True	TCustomForm
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
VertScrollBar	published		TScrollingWinControl
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl
WindowState	public	wsNormal	TCustomForm

AXBorderStyle

Specifies the style of the *TActiveForm*'s border.

property AxBorderStyle: TActiveFormBorderStyle;

Description The following are possible values for *AXBorderStyle*:

Value	Meaning
afbNone	The form has no border.
afbSingle	The form has a regular single-line border.
afbSunken	The form has a 3D look—that is, a border with a sunken edge.
afbRaised	The form has a border with a raised edge.

See also *TForm.BorderStyle* property

Events

The following table lists the events of *TActiveForm*:

Event	Visibility	Documented in
OnActivate	published	TCustomForm
OnClick	published	TControl
OnCreate	published	TCustomForm
OnDblClick	published	TControl
OnDeactivate	published	TCustomForm
OnDestroy	published	TCustomForm
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnPaint	published	TCustomForm

Methods

The following table lists the methods of *TActiveForm*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject

Method	Visibility	Documented in
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
Close	public	TCustomForm
CloseQuery	public	TCustomForm
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TActiveForm
CreateNew	public	TCustomForm
CreateParented	public	TWinControl
DefaultHandler	public	TObject
DefocusControl	public	TCustomForm
Destroy	public	TCustomForm
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
DisableAutoRange	public	TScrollingWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EnableAutoRange	public	TScrollingWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TCustomForm
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetFormImage	public	TCustomForm
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl

Method	Visibility	Documented in
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TCustomForm
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Print	public	TCustomForm
Realign	public	TWinControl
Refresh	public	TControl
Release	public	TCustomForm
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
ScrollInView	public	TScrollingWinControl
SendCancelMode	public	TCustomForm
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TCustomForm
SetFocusedControl	public	TCustomForm
SetTextBuf	public	TControl
Show	public	TCustomForm
ShowModal	public	TCustomForm
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TActiveForm*.

constructor Create(AOwner: TComponent); **override**;

Description *Create* calls and then overrides the inherited constructor to initialize the *AxBorderStyle* property.

See also *AXBorderStyle* property, *TActiveFormFactory* object

TAnimate

Unit: ComCtrls | Palette: Win32

TObject • TPersistent • TComponent • TControl • TWinControl • TAnimate

TAnimate is a wrapper for a Windows animation control.

Description Use *TAnimate* to put an animation control on a form. An animation control is a window that silently displays an Audio Video Interleaved (AVI) clip. An AVI clip is a series of bitmap frames like a movie. Although AVI clips can have sound, animation controls only work with silent AVI clips. *TAnimate* works with uncompressed AVI files or AVI clips compressed using run-length encoding (RLE).

TAnimate can obtain its AVI clip from an AVI resource, an AVI file, or, if the application is running under Windows 95 or NT 4.0, from Shell32.dll.

See also *TImage* object, *TMediaPlayer* object

Properties

The following table lists the properties of *TAnimate*:

Property	Visibility	Default	Documented in
Active	published		TAnimate
Align	published	alNone	TControl
AutoSize	published	True	TAnimate
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Center	published	True	TAnimate
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
CommonAVI	published	aviNone	TAnimate
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent

Property	Visibility	Default	Documented in
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
FileName	published		TAnimate
FrameCount	public, read-only		TAnimate
FrameHeight	public, read-only		TAnimate
FrameWidth	public, read-only		TAnimate
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Open	public		TAnimate
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	True	TControl
ParentWindow	public		TWinControl
Repetitions	published	0	TAnimate
ResHandle	public		TAnimate
ResID	public		TAnimate
ResName	public		TAnimate
ShowHint	public		TControl
Showing	public, read-only		TWinControl
StartFrame	published	1	TAnimate
StopFrame	published	0	TAnimate
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Timers	published	False	TAnimate
Top	published		TControl
Transparent	published	True	TAnimate
VCLComObject	public		TComponent
Visible	published	True	TControl

Property	Visibility	Default	Documented in
Width	published		TControl
WindowProc	public		TControl

Active

Active indicates whether the animation control is playing the AVI clip.

property Active: Boolean;

Description Set *Active* to *True* to cycle through the frames specified by the *StartFrame* and *StopFrame* properties by the number of iterations specified in the *Repetitions* property. Set *Active* to *False* to interrupt the animation control when it is playing the AVI clip.

The animation control must be *Open* before it can be *Active*.

See also *OnStart* event, *OnStop* event, *Open* property, *Play* method, *Repetitions* property, *StartFrame* property, *Stop* method, *StopFrame* property

AutoSize

AutoSize determines whether the animation control adjusts its size to the frames in the AVI clip.

property AutoSize: Boolean;

Description Set *AutoSize* to *True* to have the animation control automatically size itself to the size of the frames in the AVI clip. Set *AutoSize* to *False* to make the animation control fixed in size. When *AutoSize* is *False*, the frames of the AVI clip may not take up the entire client area of the control, or they may appear clipped if they are larger than the client area of the control.

Note If the value of the *Align* property is not *alNone*, setting *AutoSize* to *True* only resizes the sides of the animation control that are not sized to match the parent control.

See also *Align* property, *Center* property, *ClientRect* property, *FrameHeight* property, *FrameWidth* property

Center

Center determines whether the frames of the AVI clip are centered in the animation control.

property Center: Boolean;

Description Use *Center* to position the frames of the AVI clip in the animation control when *FrameHeight* is not the same as *ClientHeight* or *FrameWidth* is not the same as *ClientWidth*. When *Center* is *True*, the frames are centered both horizontally and vertically in the client area of the control. When *Center* is

False, the top left corner of the frames is positioned at the top left corner of the client area.

See also *AutoSize* property, *ClientHeight* property, *ClientWidth* property, *FrameHeight* property, *FrameWidth* property

CommonAVI

CommonAVI indicates whether the animation control displays a one of the common Windows AVI clips provided in Shell32.dll.

```
type TCommonAVI = (aviNone, aviFindFolder, aviFindFile, aviFindComputer, aviCopyFiles,
  aviCopyFile, aviRecycleFile, aviEmptyRecycle, aviDeleteFile);

property CommonAVI: TCommonAVI;
```

Description Set *CommonAVI* to display one of the common Windows AVI clips in the animation control. The common Windows AVI clips appear in the Windows user interface when the user attempts common file actions. *CommonAVI* is *aviNone* when the animation control displays an AVI clip that was specified in some other manner.

Setting *CommonAVI* automatically sets the *FileName* property to an empty string, *ResHandle* and *ResID* to 0, and *ResName* to an empty string.

Note *CommonAVI* cannot be used when running under NT 3.51.

See also *FileName* property, *ResHandle* property, *ResID* property, *ResName* property

FileName

FileName is the name of the file that contains the AVI clip displayed by the animation control.

```
type TFileName = string;

property FileName: TFileName;
```

Description Set *FileName* to use the animation control to display the AVI clip stored in a file. *FileName* must be the name of a file containing a silent AVI clip.

Note Setting *FileName* automatically sets the *CommonAVI* property to *aviNone*, *ResHandle* and *ResID* to 0, and *ResName* to an empty string.

See also *CommonAVI* property, *ResHandle* property, *ResID* property, *ResName* property

FrameCount

FrameCount is the total number of frames in the AVI clip.

```
property FrameCount: Integer;
```

Description Read *FrameCount* to determine the number of frames that can be displayed by the animation control. The animation control can play all the frames in the AVI clip, or run through a subset of those frames.

See also *Play* method, *StartFrame* property, *StopFrame* property

FrameHeight

FrameHeight is the height, in pixels, of each image in the AVI clip.

```
property FrameHeight: Integer;
```

Description Read *FrameHeight* to determine the height required to fully display the frames of the AVI clip. If *AutoSize* is *True*, the animation control automatically sizes itself to the value of *FrameHeight*.

See also *AutoSize* property, *ClientHeight* property, *FrameWidth* property

FrameWidth

FrameWidth is the width, in pixels, of each image in the AVI clip.

```
property FrameWidth: Integer;
```

Description Read *FrameWidth* to determine the width required to fully display the frames of the AVI clip. If *AutoSize* is *True*, the animation control automatically resizes itself to the value of *FrameWidth*.

See also *AutoSize* property, *ClientWidth* property, *FrameHeight* property

Open

Open indicates whether the AVI clip is loaded into memory.

```
property Open: Boolean;
```

Description Read *Open* to determine whether the AVI clip is loaded and ready for playing. When the AVI clip is specified using the *CommonAVI*, *FileName*, *ResName*, or *ResID* property, the animation control is opened automatically. When *Open* is *True*, the animation control displays the frame specified by the *StartFrame* property until the clip is played using the *Active* property or *Play* method.

Set *Open* to *False* to free the resources used by the AVI clip. When the animation control is not open, none of the frames can be shown.

Note When *Open* is changed from *False* to *True*, the *StartFrame* property is reset to 1 and the *StopFrame* property is reset to *FrameCount*.

See also *Active* property, *CommonAVI* property, *FileName* property, *FrameCount* property, *OnClose* event, *OnOpen* event, *Play* method, *ResHandle* property, *ResID* property, *ResName* property, *StartFrame* property, *StopFrame* property

Repetitions

Repetitions is the number of times to repeat the sequence of frames.

```
property Repetitions: Integer;
```

Description Set *Repetitions* to specify the number of times the sequence of frames in the AVI clip is repeated when the *Active* property is set to *True*. After the sequence is repeated by the specified number of times, the value of *Active* becomes *False* again.

To show the sequence a single time, set *Repetitions* to 1. Set *Repetitions* to 0 to indicate that the sequence of frames should repeat indefinitely. When *Repetitions* is 0, use the *Active* property or *Stop* method to interrupt the animation.

Note The *Play* method overrides *Repetitions* with the value passed as the *Count* parameter.

See also *Active* property, *Play* method, *Stop* method

ResHandle

ResHandle is the Windows handle for the module that contains the AVI clip as a resource.

```
type THandle = Integer;
property ResHandle: THandle;
```

Description Set *ResHandle* at runtime to the instance handle or module handle of the module that includes the animation resource. After setting *ResHandle*, set the *ResID* or *ResName* property to specify which resource in the indicated module is the AVI clip that should be displayed by the animation control.

If a value is not specified for *ResHandle*, the animate control uses the instance of the application's main executable.

Note Setting *ResHandle* automatically sets the *CommonAVI* property to *aviNone* and the *FileName* property to an empty string.

See also *CommonAVI* property, *FileName* property, *HInstance* variable, *MainInstance* variable, *ResID* property, *ResName* property

ResID

ResID is the resource ID of the AVI clip.

```
property ResID: Integer;
```

Description Set *ResID* to identify a resource that should be used as the AVI clip displayed by the animation control. Use *ResHandle* to indicate the module that contains the resource identified by *ResID*.

If the resource file identifies the AVI clip by name rather than number, use the *ResName* property instead.

Note Setting *ResID* automatically sets the *CommonAVI* property to *aviNone*, the *FileName* property to an empty string, and the *ResName* property to an empty string.

See also *CommonAVI* property, *FileName* property, *ResHandle* property, *ResName* property

ResName

ResName is the resource name of the AVI clip.

```
property ResName: string;
```

Description Set *ResName* to identify a resource that should be used as the AVI clip displayed by the animation control. Use *ResHandle* to indicate the module that contains the resource identified by *ResName*.

If the resource file identifies the AVI clip by number rather than number, use the *ResID* property instead.

Note Setting *ResName* automatically sets the *CommonAVI* property to *aviNone*, the *FileName* property to an empty string, and the *ResID* property to 0.

See also *CommonAVI* property, *FileName* property, *ResHandle* property, *ResID* property

StartFrame

StartFrame is the first frame displayed when the animation control is activated.

```
property StartFrame: SmallInt;
```

Description Set *StartFrame* to specify the first frame in the AVI clip that is shown when the *Active* property is set to *True*. When *StartFrame* is 1, the clip is played from the very beginning, when *StartFrame* is 2, the clip begins with the second frame, and so on. If *Repetitions* is not 1, after the sequence of frames is played, it starts over at the frame specified by *StartFrame* (which may not be the first frame in the clip).

When the animation control is *Open* but not *Active*, it displays the frame specified by *StartFrame*.

Note The *Play* method overrides *StartFrame* with the value passed as the *FromFrame* parameter.

See also *Active* property, *FrameCount* property, *Play* method, *Repetitions* property, *Seek* method, *StopFrame* property

StopFrame

StopFrame is the last frame displayed when the animation control is activated.

property StopFrame: SmallInt;

Description Set *StopFrame* to specify the last frame in the AVI clip that is shown every time the animation control displays its sequence of frames. When *StopFrame* is 0, the clip is played to the very end. Otherwise, *StopFrame* is the number of the frame in the sequence. When the *Active* property is set to *True*, the animation control displays the sequence of frames from *StartFrame* to *StopFrame*, repeating this sequence by the number of iterations specified by the *Repetitions* property.

Note The *Play* method overrides *StopFrame* with the value passed as the *ToFrame* parameter.

See also *Active* property, *FrameCount* property, *Play* method, *Repetitions* property, *StartFrame* property

Timers

Timers specifies whether the frames are displayed using a timer.

property Timers: Boolean;

Description Set *Timers* to specify whether the frames in the AVI clip should be changed in response to timer messages, or whether the sequence progresses asynchronously in a separate execution thread. When *Timers* is *True*, the frames progress in response to timer messages. This is useful for synchronizing the animation sequence with other actions such as playing a sound track. When *Timers* is *False*, the AVI clip is played in a separate thread.

See also *OnStart* event, *TMediaPlayer* object

Transparent

Transparent specifies whether the background color of the frame matches the *Color* property.

property Transparent: Boolean;

Description Set *Transparent* to *True* to replace the background color stored in the AVI file with the value of the *Color* property, or the parent control's color if the value of *ParentColor* is *True*. Set *Transparent* to *False* to display the background color stored in the AVI clip.

See also *Color* property, *ParentColor* property

Events

The following table lists the events of *TAnimate*:

Event	Visibility	Documented in
OnClose	published	TAnimate
OnOpen	published	TAnimate
OnStart	published	TAnimate
OnStop	published	TAnimate

OnClose

OnClose occurs immediately after the animation control is closed.

```
type TNotifyEvent = procedure(Sender: TObject) of object;
property OnClose: TNotifyEvent;
```

Description Write an *OnClose* event handler to take specific action when the *Open* property is set to *False*. For example, if the animation control is used to cycle through a series of AVI clips, the clip could be changed in an *OnClose* event handler.

To respond when the clip stops playing, but the animation control remains open, use the *OnStop* event.

See also *OnOpen* event, *OnStop* event, *Open* property

OnOpen

OnOpen occurs immediately after the animation control is opened.

```
type TNotifyEvent = procedure(Sender: TObject) of object;
property OnOpen: TNotifyEvent;
```

Description Write an *OnOpen* event handler to take specific action when the *Open* property changes from *False* to *True*. This is a good opportunity to set the *Repetitions*, *StartFrame*, and *StopFrame* properties before the AVI clip is activated, or to seek to a particular frame.

See also *Active* property, *OnClose* event, *OnStart* event, *Open* property, *Repetitions* property, *Seek* method, *StartFrame* property, *StopFrame* property

OnStart

OnStart occurs immediately after the animation control starts playing.

```
type TNotifyEvent = procedure(Sender: TObject) of object;
property OnStart: TNotifyEvent;
```

Description Write an *OnStart* event handler to take specific action when the AVI clip is played. Setting the *Active* property to *True* or calling the *Play* method triggers an *OnStart* event. *OnStart* is useful for launching other actions that should be synchronized with the playing of the animation sequence.

See also *Active* property, *OnOpen* event, *OnStop* event, *Play* method, *Timers* property

OnStop

OnStop occurs when the animation control finishes playing.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnStop: TNotifyEvent;
```

Description Write an *OnStop* event handler to take specific action when the AVI clip finishes playing all its repetitions. For example, use an *OnStop* event handler to shut down any simultaneous actions that were started in an *OnStart* event handler.

See also *Active* property, *OnClose* event, *OnStart* event, *Repetitions* property, *Stop* method

Methods

The following table lists the methods of *TAnimate*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TAnimate
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent

Method	Visibility	Documented in
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Play	public	TAnimate
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl

Method	Visibility	Documented in
Repaint	public	TWinControl
Reset	public	TAnimate
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
Seek	public	TAnimate
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Stop	public	TAnimate
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates and initializes an animation control.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate an animation control at runtime. *Animation* controls placed on forms at design time are created automatically. After calling the inherited constructor, *Create* initializes the following properties:

- *AutoSize* is set to *True*.
- *ControlStyle* is set to [].
- *Transparent* is set to *True*.
- *Center* is set to *True*.
- *StartFrame* is set to 1.

See also *AutoSize* property, *Center* property, *ControlStyle* property, *StartFrame* property, *Transparent* property

Play

Play displays a sequence of frames from the AVI clip.

procedure Play(FromFrame, ToFrame: Word; Count: Integer);

Description Call *Play* to display the specified sequence of frames, repeating the sequence *Count* times. *FromFrame* is the first frame in the sequence, where 1 specifies the first frame in the clip, 2 specifies the second frame, and so on. *ToFrame* is the last frame in the sequence. The value of *ToFrame* should be greater than or equal to *FromFrame* and less than or equal to the value of the *FrameCount* property. *Count* specifies the number of times the sequence should be

repeated. If *Count* is 0, the sequence is repeated until the *Stop* method is called.

If the animation control is not *Open* when *Play* is called, *Play* sets *Open* to *True*.

Note Calling *Play* is the same as setting *StartFrame* to *FromFrame*, *StopFrame* to *ToFrame*, *Repetitions* to *Count*, and then setting *Active* to *True*.

See also *Active* property, *Open* property, *Repetitions* property, *StartFrame* property, *StopFrame* property

Reset

Reset clears the values of *StartFrame* and *StopFrame*, and displays the first frame in the AVI clip.

```
procedure Reset;
```

Description Call *Reset* to restore the animation control to a default state. *Reset* closes and reopens the animation control, generating *OnClose* and *OnOpen* events. The animation control is left open but not active.

See also *Active* property, *OnClose* event, *OnOpen* event, *Open* property, *StartFrame* property, *StopFrame* property

Seek

Seek displays a specified frame in the sequence.

```
procedure Seek(Frame: SmallInt);
```

Description Call *Seek* to move to a particular frame in the AVI clip. The *Frame* parameter specifies which frame, where 1 is the first frame in the sequence, 2 is the second frame, and so on. *Frame* must be a value less than or equal to *FrameCount*.

If the animation control is not open when *Seek* is called, *Seek* sets *Open* to *True*.

See also *FrameCount* property, *Open* property

Stop

Stop interrupts the animation control when it is playing.

```
procedure Stop;
```

Description Call *Stop* to halt the sequence of images in an active animation control. *Stop* does the same thing as setting the *Active* property to *False*.

See also *Active* property, *Play* method

TApplication

Unit: Forms

TObject • TPersistent • TComponent • TApplication

TApplication is the type used for a Delphi application.

Description *TApplication* is a class that encapsulates a windows application. The methods and properties introduced in *TApplication* reflect the fundamentals established in the Windows operating system to create, run, sustain, and destroy an application. *TApplication* thereby simplifies the interface between the user and the Windows environment when writing a Windows application. For this purpose, *TApplication* encapsulates behavior for

- Windows message processing.
- Context-sensitive online Help.
- Menu accelerator and key processing.
- Exception handling.
- Managing the fundamental parts defined by the Windows operating system for an application, such as MainWindow, WindowClass and so on.

Each Delphi application automatically declares an *Application* variable as the instance of the application. If the Delphi application is not a Web server application, this variable is of type *TApplication*. Therefore *TApplication* does not need to appear on the component palette, nor is it available in the form designer to visually manipulate; so it has no published properties. Nevertheless, some of its public properties can be set or modified at design time in the Forms and Application pages of the Project | Options dialog box.

See also *Application* variable, *Exception* object, *TComponent* object, *TControl* object, *TForm* object, *TWinControl* object

Properties

The following table lists the properties of *TApplication*:

Property	Visibility	Default	Documented in
Active	public, read-only		TApplication
AllowTesting	public		TApplication
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent

Property	Visibility	Default	Documented in
DesignInfo	public		TComponent
DialogHandle	public		TApplication
ExeName	public, read-only		TApplication
Handle	public		TApplication
HelpFile	public		TApplication
Hint	public		TApplication
HintColor	public		TApplication
HintHidePause	public		TApplication
HintPause	public		TApplication
HintShortPause	public		TApplication
Icon	public		TApplication
MainForm	public, read-only		TApplication
Name	published		TComponent
Owner	public, read-only		TComponent
ShowHint	public		TApplication
ShowMainForm	public		TApplication
Tag	published	0	TComponent
Terminated	public, read-only		TApplication
Title	public		TApplication
UpdateFormatSettings	public		TApplication
UpdateMetricSettings	public		TApplication
VCLComObject	public		TComponent

Active

Active specifies whether the application is active and has focus.

property Active: Boolean;

Description *Active* is read-only. *Active* is *True* while the application is active and *False* if it is not. *Active* is set to *True* in the constructor.

An application is active if the form or application has focus. An application becomes inactive when a window from a different application is about to become activated. When the application closes, *Active* is set to *False* in the destructor.

See also *OnActivate* event, *OnDeactivate* event

AllowTesting

AllowTesting is for internal use only.

property AllowTesting: Boolean

Description Do not use *AllowTesting;* it is for internal use only.

DialogHandle

DialogHandle provides a mechanism for using non-Delphi dialog boxes in a Delphi application.

property DialogHandle: HWnd;

Description Use *DialogHandle* when creating a modeless dialog box that was created using the CreateDialog API function, and that needs to see messages from the application's message loop in order to operate.

To do this, assign the handle of the modeless dialog box to *DialogHandle* when it receives an activation message (WM_NCACTIVATE) and set *DialogHandle* to zero when the dialog box receives a deactivation message.

Example Here is part of the window procedure for Delphi's encapsulation of the Windows search-and-replace common dialog box:

```
function SearchAndReplaceWndProc(Wnd: HWnd; Msg, WParam: Word; LParam: Longint):
Longint;
begin
  try
    case Msg of
      WM_DESTROY: Application.DialogHandle := 0;{ clear handle on destroy }
      WM_NCACTIVATE:
        if Bool(WParam) then{ if it's activating... }
          Application.DialogHandle := Wnd{ ...assign the handle... }
        else Application.DialogHandle := 0;{ ...otherwise clear it }
      :
    end;
    Result := CallDefDialogProc;
  except
    Application.HandleException(nil);
  end;
end;
```

See also *Handle* property

ExeName

ExeName contains the file name of the executable application including path information.

property ExeName: string;

Description *ExeName* is a read-only property. Use *ExeName* to obtain the name of the executable file for the application.

The name of the application is the root name of the project file with an .EXE extension. By default this name is PROJECT1.EXE. To change the *ExeName*, save the project by the desired new root name and rebuild the application. *ExeName* will reflect the change to the project filename.

See also *GetModuleFileName* function, *ParamStr* function, *Title* property

Handle

Handle provides access to the window handle of the main form (window) of the application.

property Handle: HWND;

Description Use *Handle* when calling a Windows API functions that require a parent window handle. For example, a DLL that displays its own top-level pop-up windows will need a parent window to display its windows in the application. Using the *Application.Handle* makes such windows part of the application, so that they will be minimized, restored, enabled and disabled with the application.

Note When writing a DLL that uses VCL forms, assign the window handle of the host EXE's main window to the DLL's *Application.Handle* property. This makes the DLL's form part of the host application. Never assign to *Application.Handle* in an EXE.

See also *DialogHandle* property, *MainForm* property

HelpFile

HelpFile specifies the name of the file the application uses to display Help.

property HelpFile: string;

Description Use *HelpFile* for applications that have a Help file to use with the Windows Help system. Windows displays the file specified by the *HelpFile* property. Either assign a filename to the *HelpFile* property at runtime, or specify a Help File on the Application page of the Project | Options dialog box at design time.

By default, *HelpFile* is a null string, and the application's *Help* method ignores attempts to display Help. If *HelpFile* contains anything, the *HelpContext* method passes it to the Windows Help system as the name of the file to use for Help.

Note The active form's Help file will be displayed rather than the application's Help file, if it has been assigned.

See also *HelpCommand* method, *HelpContext* method, *HelpJump* method, *TCustomForm.HelpFile* property

Hint

Hint specifies the text string that appears in the Help Hint box.

property Hint: string;

Description *TApplication*'s *Hint* property is assigned the *Hint* value of the control or menu item that the mouse is currently moving over. It can also be assigned a string

value that informs the user of some action, error, or other type of information. Therefore, use *Hint* either to

- Transfer hints from controls to another area of display, such as a status bar, using the *OnHint* event handler. This reads the *Hint* property.
- Indicate the status of the application while it is processing some action. This sets the *Hint* property.

Help Hints appear when the *OnHint* event occurs. Therefore, if *TApplication*'s *Hint* property is assigned to the status bar's caption, for example, the caption of the status bar control displays the current string value of the *Hint* property during an *OnHint* event.

There are two parts to the *Hint* string: short and long. Short hints are used by pop-up tool tips. Long hints are used by the status bar, separated by the | character. Use the global functions *GetShortHint* and *GetLongHint*, from the *Controls* unit, to extract the long and short hints from a hint string.

Note When setting *Hint* to a message informing the user of an event occurring in the application, remember that, by default, the *Hint* string is reset to a control's *Hint* when the mouse cursor moves over a control.

See also *GetLongHint* function, *GetShortHint* function, *HintColor* property, *OnHint* event, *ShowHint* property

HintColor

HintColor determines the color of the hint boxes for the Help Hints for the application.

```
property Color: TColor;
```

Description Use *HintColor* to specify the hint box color. A default color value of *clInfoBk* is set for the *HintColor* property in the constructor when the application is created. To change the *HintColor* assign it a new *TColor* value at runtime.

See also *Hint* property, *TColor* type

HintHidePause

HintHidePause specifies the time interval to wait before hiding the Help Hint if the mouse has not moved from the control or menu item.

```
property HintHidePause: Integer;
```

Description Use *HintHidePause* to specify a wait time in milliseconds, which is different from the default value of 2500 ms or 2 1/2 seconds that is set in the constructor. The Help Hint for the control or menu item is specified in the *Hint* property.

Note The default is a predefined **const** value that is several times longer than *HintPause*.

See also *Hint* property, *HintPause* property, *HintShortPause* property

HintPause

HintPause specifies the time interval that passes before the control's Help Hint appears when the user places the mouse pointer on a control or menu item.

```
property HintPause: Integer;
```

Description Use *HintPause* to change the default pause time of 500 ms or 1/2 second which is set in the constructor. When assigning a value to *HintPause* specify the interval in milliseconds. The Help Hint displayed at the end of the pause is specified in the *Hint* property.

See also *HideHintPause* property, *Hint* property, *HintShortPause* property

HintShortPause

HintShortPause specifies the pause to wait before bringing up a hint if a hint has already been shown.

```
property HintShortPause: Integer;
```

Description Use *HintShortPause* to reduce the flicker when moving the mouse quickly over a set of buttons that all have help hints on. Specify a value in milliseconds that is different from the default value of 50 ms which is set in the constructor.

See also *HideHintPause* property, *Hint* property, *HintPause* property

Icon

Icon determines which icon represents the application on the Windows task bar.

```
property Icon: TIcon;
```

Description Use *Icon* to assign a specific icon to represent the application. Set the *Icon* property by either

- Assigning it a *TIcon* at runtime.
- Loading an icon on the Application pages of the Project | Options dialog box at design time.

Icon represents the application when it is minimized and appears either on the Windows desktop or on the task bar in Windows 95.

See also *Minimize* method, *OnMinimized* event, *Restore* method, *Title* property

MainForm

MainForm identifies which form in the application is the main window.

```
property MainForm: TForm;
```

Description Use *MainForm* to specify a form, different from the default, as the main window of the application. This value is set to be the first form created via the *CreateForm* method. When a new project is created, Form1 automatically becomes the value of the *MainForm* property. To assign a different form to the *MainForm* property, use the Forms page of the Project | Options dialog box at design time. *MainForm* cannot be modified at runtime because it is read-only.

The main form is the first form created in the main body of the application. When the main form closes, the application terminates.

See also *Create* method, *CreateForm* method, *Handle* property, *Terminate* method

ShowHint

Showing determines whether Help Hints are enabled or disabled for the entire application.

```
property ShowHint: Boolean;
```

Description Use *ShowHints* to choose when to display Help Hints. If *ShowHint* is *True*, Help Hints are enabled; if *ShowHint* is *False*, Help Hints are disabled. The default value is *True*. Help Hints are specified in the *Hint* property.

Setting *ShowHint* for the application to *False* disables all Help Hints, regardless of the value of the *ShowHint* properties for individual controls.

See also *Hint* property, *OnHint* event, *TControl.ParentShowHint* property, *TControl.ShowHint* property

ShowMainForm

ShowMainWindow determines whether the application shows its main form on startup.

```
property ShowMainForm: Boolean;
```

Description Use *ShowMainForm* to control whether and when the application shows its main form.

The default value of *True* is set in the constructor. By default the main window shows. The *MainForm* property specifies which form is the main form.

To hide the main form at startup, set *ShowMainForm* to *False* in the main project file before the call to *Application.Run*, and make sure that the *Visible* property of the form is also *False*. This is mostly useful for implementing OLE automation servers, to hide the server's main form when started for automation purposes.

See also *Create* method, *MainForm* property, *Run* method, *Visible* property

Terminated

Terminated reports whether the application has received the Windows WM_QUIT message, which then terminates the application.

property Terminated: Boolean;

Description *Terminated* is read-only. It is used when calling the *ProcessMessages* method so that the application doesn't attempt to process Windows messages after the application has quit. *ProcessMessage* sets *Terminated* to *True* when it receives a WM_QUIT message.

A Delphi application usually receives a WM_QUIT message because the main window of the application has closed, or because the *Terminate* method has been called.

For applications using calculation-intensive loops, call *Application.ProcessMessages* periodically, and also check *Application.Terminated* to determine whether or not to abort the calculation so that the application can terminate.

See also *ProcessMessages* method, *Terminate* method

Title

Title contains the text that appears below the icon representing the application when it is minimized.

property Title: string;

Description Use the *Title* property to modify the name of this text string appearing below the icon of the application. The default value is taken from the DLL/EXE filename at runtime.

Set the *Title* at runtime, or enter the value of the *Title* property on the Application page of the Project | Options dialog box.

See also *ExeName* property, *Icon* property

UpdateFormatSettings

UpdateFormatSettings specifies whether the format settings are updated automatically when the user alters the system configuration.

property UpdateFormatSettings: Boolean;

Description Use *UpdateFormatSettings* to control automatic updating of format settings. The default of *True* is set in the constructor. *UpdateFormatSettings* is checked when the application receives a WM_WININICHANGE message.

It is recommended to use the default format settings, which are initialized to the Windows local settings. However, to prevent the settings from changing during the execution of the application when altering the format settings in a Delphi application, *UpdateFormatSettings* can be set to *False*.

UpdateMetricSettings

UpdateMetricSettings specifies whether the settings used for the hint window font and the icon title font are updated.

```
property UpdateMetricSettings: Boolean;
```

Description Use *UpdateMetricSettings* to specify whether or not the hint window font and the icon title font on the desk top should be updated to reflect the system settings when those settings are changed.

The initial value of *True* is set in the constructor.

Events

The following table lists the events of *TApplication*:

Event	Visibility	Documented in
OnActivate	public	TApplication
OnDeactivate	public	TApplication
OnException	public	TApplication
OnHelp	public	TApplication
OnHint	public	TApplication
OnIdle	public	TApplication
OnMessage	public	TApplication
OnMinimize	public	TApplication
OnRestore	public	TApplication
OnShowHint	public	TApplication

OnActivate

OnActivate occurs when an application becomes active.

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

```
property OnActivate: TNotifyEvent;
```

Description Use *OnActivate* to write an event handler to perform special processing when the application becomes activated.

An application becomes active when it is initially run or when focus moves from another Windows application back to any window of the application.

See also *Active* property, *OnDeactivate* event

OnDeactivate

OnDeactivate occurs when an application becomes inactive.

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

```
property OnDeactivate: TNotifyEvent;
```

Description Use the *OnDeactivate* event to specify any special processing that should occur immediately before the application becomes deactivated. The *OnDeactivate* event occurs when the user switches from the application to another Windows application.

See also *Active* property, *OnActivate* event

OnException

OnException occurs when an unhandled exception occurs in the application.

```
type TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;

property OnException: TExceptionEvent;
```

Description Use *OnException* to change the default behavior that occurs during an unhandled exception in the application. The *OnException* event handler is called automatically in the *HandleException* method.

OnException only handles exceptions that occur during message processing. Exceptions that occur before or after the *Application.Run* do not go through *Application.HandleException* or *OnException*.

If an exception passes through the try blocks in the application code, the application automatically calls the *HandleException* method, which displays a dialog box indicating an error occurred. Unless the exception object is *EAbort*, *HandleException* calls the *OnException* handler, if one exists, otherwise it calls *ShowException* to display a message dialog box indicating an error occurred.

The *TExceptionEvent* type is the type of the *OnException* event. It points to a method that handles exceptions in the application. The *Sender* parameter is the object that raised the exception, and *E* is the exception object.

See also *HandleException* method, *ShowException* method

OnHelp

OnHelp occurs when the application receives a request for help.

```
type THelpEvent = function (Command: Word; Data: Longint; var CallHelp: Boolean): Boolean
    of object;

property OnHelp: THelpEvent;
```

Description Use *OnHelp* to write an event handler to perform special processing when help is requested. The *HelpContext* and the *HelpJump* methods automatically trigger the *OnHelp* event.

Set *CallHelp* to *True* to have the VCL call *WinHelp* after the event. Set *CallHelp* to *False* to prevent the VCL from calling *WinHelp*.

All application help methods go through *OnHelp*. Only if *OnHelp*'s *CallHelp* parameter returns *True*, or if *OnHelp* is not assigned, is *WinHelp* called.

To find the possible values of the *Command* and *Data* parameters, search for WinHelp in the Win32 Developer's Reference Help (Win32.HLP) file, which explains the WinHelp API (application programming interface). The possible values for the *Data* parameter depend upon the value of the *Command* parameter.

The return value is *True* if the function succeeds, *False* if the function fails.

See also *HelpCommand* method, *HelpContext* method, *HelpFile* property, *HelpJump* method

OnHint

OnHint occurs when the mouse pointer moves over a control or menu item that can display a Help Hint.

```
type TNotifyEvent = procedure (Sender: TObject) of object;

property OnHint: TNotifyEvent;
```

Description Use *OnHint* to write an event handler to perform special processing when the *OnHint* event occurs. *OnHint* occurs when the user positions the mouse pointer over a control with a *Hint* property value other than an empty string ('').

A common use of the *OnHint* event is to display the value of a control or menu item's *Hint* property as the caption of a panel control (*TStatusBar*), thereby using the panel as a status bar. The *Hint* property can specify a Help Hint and a usually longer hint that appears elsewhere when the *OnHint* event occurs.

Example This example uses a status bar component, a menu, and an edit box on a form. Include a value for the *Hint* property for each menu item in the menu. Also, specify a value for the *Hint* property of the edit box.

To write the *OnHint* event handler, create a method of the TForm1 object and give it an appropriate name, such as *DisplayHint*. Write the method in the implementation part of the unit, declare the method in the TForm1 type declaration in the public section.

In the *DisplayHint* method assign the *Hint* property of the application to the *Caption* property of the panel component.

Assign this new method as the method used by the *OnHint* event. In the example this is done in the form's *OnCreate* event handler.

This code shows the complete type declaration, the new method, and the *OnCreate* event handler. When the user runs the application and positions the cursor over the edit box or a menu item on the menu, the specified hint appears as the caption of the panel at the bottom of the form:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Statubar1: TStatusBar;
    Edit1: TEdit;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    procedure DisplayHint(Sender: TObject);
  end;
var
  Form1: TForm1;

implementation
{$R *.FRM}

procedure TForm1.DisplayHint(Sender: TObject);
begin
  Statubar1.Caption := GetLongHint(Application.Hint);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := DisplayHint;
end;
```

See also *GetLongHint* function, *GetShortHint* function, *Hint* property

OnIdle

OnIdle occurs when an application becomes idle.

type TIdleEvent = **procedure** (Sender: TObject; **var** Done: Boolean) **of object**;

property OnIdle: TIdleEvent

Description Use *OnIdle* to write an event handler to perform special processing when an application is idle. An application is idle when it is processing code, for example, or when it is waiting for input from the user.

OnIdle is called only once, as the application transitions into an idle state. It is not called continuously unless *Done* is set to *False*. If *Done* is *False*, *WaitMessage* is not called. Applications that set *Done* to *False* consume an inordinate amount of CPU time that affects overall system performance.

The *TIdleEvent* type is the type of the *OnIdle* event. It points to a method that runs when the application is idle. The Boolean parameter *Done* is *True* by default. When Done is *True*, the Windows API *WaitMessage* function is called when *OnIdle* returns. *WaitMessage* yields control to other applications until a new message appears in the message queue of the application.

OnMessage

OnMessage processes Windows messages that an application receives.

```
type TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object;
property OnMessage: TMessageEvent;
```

Description Use *OnMessage* to trap any or all Windows messages posted to all windows in the application. The *OnMessage* event occurs when an application receives a Windows message. Creating an *OnMessage* event handler in an application allows other handlers to respond to messages other than those declared in the events for *TApplication*. If the application doesn't have a specific handler for an incoming message, the message is dispatched to the window for which it was intended, and Windows handles the message.

The *TMessageEvent* type is the type of the *OnMessage* event. It points to a method that handles the processing of incoming Windows messages. The *Msg* parameter identifies the Windows message, and the *Handled* parameter determines whether the message is handled or not. Set *Handled* to *True*, if the message has been completely handled, to prevent normal processing of the message.

Note *OnMessage* only receives messages that are posted to the message queue, not those sent directly with the Windows API *SendMessage* function.

Caution Thousands of messages per second flow though this event. Be careful when coding the handler since it can affect the performance of the entire application.

See also *HandleMessage* method, *ProcessMessages* method

OnMinimize

OnMinimize occurs when an application is minimized.

```
type TNotifyEvent = procedure (Sender: TObject) of object;
property OnMinimize: TNotifyEvent;
```

Description Use *OnMinimize* to write an event handler to perform special processing when the application is minimized. The application is minimized, either because the user minimizes the main window, or because of a call to the *Minimize* method. The *Icon* property determines the icon that represents the minimized application.

See also *Icon* property, *Minimize* method, *OnRestore* event, *Restore* method

OnRestore

OnRestore occurs when the previously minimized application is restored to its normal size.

```
type TNotifyEvent = procedure (Sender: TObject) of object;
property OnRestore: TNotifyEvent;
```

Description Use *OnRestore* to write an event handler to perform special processing when the application is restored from a minimized state in which it appears as an

icon. An application is restored either because the user restores the application, or because the application calls the *Restore* method.

Note Don't confuse restoring an application with restoring a form or window to its original size. To minimize, maximize, and restore a window or form, change the value of the *WindowState* property.

See also *Icon* property, *Minimize* method, *Restore* method

OnShowHint

OnShowHint occurs when the application is about to display a hint window for a Help Hint for a particular control.

```
THintInfo = record
  HintControl: TControl;
  HintPos: TPoint;
  HintMaxWidth: Integer;
  HintColor: TColor;
  CursorRect: TRect;
  CursorPos: TPoint;
  ReshowTimeout: Integer;
  HideTimeout: Integer;
  HintStr: string;
 end;

type TShowHintEvent = procedure (var HintStr: string; var CanShow: Boolean; var HintInfo:
   THintInfo) of object;

property OnShowHint: TShowHintEvent;
```

Description Use *OnShowHint* to write an event handler that changes the appearance and behavior of Help Hints.

The *TShowHintEvent* is the type of the *OnShowHint* event. It points to a method that displays a Help Hint for a control. The *HintStr* parameter sets the text of the Help Hint. To obtain the text of a hint for a particular control, call the *GetLongHint* or *GetShortHint* function, assigning the result to *HintStr*. To change the text, change the contents of this string.

Use the *CanShow* variable to permit or prevent the Help Hint from displaying. If *CanShow* is *True*, the Help Hint displays. If it is *False*, the Help Hint does not appear.

The *HintInfo* parameter is a record that contains information about the appearance and behavior of the Help window.

The *HintColor* field of the record contains the name of the control for which hint processing is occurring.

The *HintPos* field determines the default position in screen coordinates of the top-left corner of the hint window. Change where the window appears by changing this value.

The *HintMaxWidth* field determines the maximum width of the hint window before word wrapping begins. By default, the value is the width of the screen (*Screen.Width*).

The *CursorRect* field determines the rectangle the user's mouse pointer must be in for the hint window to appear. The default value for *CursorRect* is the client rectangle of the control. An application can change this value so that a single control can divided into several hint regions. When the user moves the mouse pointer moves outside the rectangle, the hint window disappears.

The *CursorPos* field contains the location of the mouse pointer within the control.

The *ReshowTimeout* field indicates how long the hint system should wait before asking about the hint status again. By default, this field is zero, indicating that the hint status will not change. Setting it to a non-zero value will cause the hint to act, after the requested milliseconds have elapsed, as if the user moved the mouse outside the hint rectangle and back in. This can be used to either put off hint processing for a period of time, or to allow the hint to be updated periodically.

The *HideTimeout* field indicates the number of milliseconds to show the hint. It is set to *Application.HintHidePause* by default.

The *HintStr* field contains the string to be displayed in the hint window. This allows the *OnHint* event to modify the contents of the hint before it is displayed. By default it contains the value of the following expression: *GetShortHint*(*Application.Hint*).

See also *GetLongHint* function, *GetShortHint* function, *HintHidePause* property, *OnHint* event

Methods

The following table lists the methods of *TApplication*:

Method	Visibility	Documented in
Assign	public	TPersistent
BringToFront	public	TApplication
CancelHint	public	TApplication
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ControlDestroyed	public	TApplication
Create	public	TApplication
CreateForm	public	TApplication

Method	Visibility	Documented in
CreateHandle	public	TApplication
DefaultHandler	public	TObject
Destroy	public	TApplication
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HandleException	public	TApplication
HandleMessage	public	TApplication
HasParent	public	TComponent
HelpCommand	public	TApplication
HelpContext	public	TApplication
HelpJump	public	TApplication
HideHint	public	TApplication
HintMouseMessage	public	TApplication
HookMainWindow	public	TApplication
InheritsFrom	public	TObject
Initialize	public	TApplication
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MessageBox	public	TApplication
MethodAddress	public	TObject
MethodName	public	TObject
Minimize	public	TApplication
NewInstance	public	TObject
NormalizeTopMosts	public	TApplication
ProcessMessages	public	TApplication
RemoveComponent	public	TComponent
Restore	public	TApplication
RestoreTopMosts	public	TApplication
Run	public	TApplication

Method	Visibility	Documented in
SafeCallException	public	TComponent
ShowException	public	TApplication
Terminate	public	TApplication
UnhookMainWindow	public	TApplication

BringToFront

BringToFront sets the last active window as the topmost window on the desktop above all other applications.

```
procedure BringToFront;
```

Description Use *BringToFront* to find the window owned by the *MainForm* that was last active and bring it to the foreground. *BringToFront* also tests to see if the window is visible and enabled before making it the topmost window.

See also *TControl.BringToFront* method, *MainForm* method

CancelHint

CancelHint cancels the displaying of a hint for a control.

```
procedure CancelHint;
```

Description Applications rarely, if ever, call *CancelHint* directly. *CancelHint* is used internally in the VCL to cancel the displaying of hints at appropriate times, such as when a window is not valid or when the application closes. *CancelHint* calls a series of routines that perform a cleanup on the display and timing issues of the hint window for a control.

See also *HideHint* method, *Hint* property

ControlDestroyed

ControlDestroyed is used internally to notify the application that a control has been destroyed.

```
procedure ControlDestroyed(Control: TControl);
```

Description *ControlDestroyed* is for internal use and is called automatically by *TControl*. There is no reason to call it directly.

Note *ControlDestroyed* notifies the application that the control passed as the *Control* parameter has been destroyed, so that the application pointers, such as *MainForm* and *ActiveForm*, can be updated to point to a valid control object or to **nil**.

See also *Destroy* method, *MainForm* property, *TScreen.ActiveForm* property

Create

Create creates a new application.

```
constructor Create(AOwner: TComponent); override;
```

Description Applications do not call *Create* directly. Each Delphi application automatically creates an instance of an application object.

For *TApplication* the *Create* method sets default values for some of its properties. It also prepares the application to run by setting up, for example, hint windows, the application icon, the main window handle, the HInstance, lists for managing child windows of the main window (form) and so on.

See also *CreateForm* method, *CreateHandle* method, *Destroy* method, *Handle* property, *MainForm* property, *Run* method

CreateForm

CreateForm creates a new form.

```
procedure CreateForm(FormClass: TFormClass; var Reference);
```

Description Applications always call *CreateForm*. However, Delphi programmers rarely write code that necessitates calling *CreateForm* directly.

A Delphi project typically contains one or more calls to *CreateForm* in the project's main statement part and handles form creation automatically when using the form designer. An application might call *CreateFrom* to dynamically create a form at runtime.

CreateForm creates a new form of the type specified by the *FormClass* parameter and assigns it to the variable given by the *Reference* parameter. The owner of the new form is the Application object. By default the form created by the first call to *CreateForm* in a project becomes the project's main form.

See also *Create* method, *CreateHandle* method, *Handle* method, *MainForm* property, *Run* method

CreateHandle

CreateHandle generates a handle for the *Handle* property of the application if it does not already have one.

```
procedure CreateHandle;
```

Description Do not call *CreateHandle* directly. In EXEs *TApplication.Create* calls *CreateHandle*. In DLLs it does not.

CreateHandle is mainly an encapsulation of much the Windows API function *WinMain*. For example, *CreateHandle* registers the Window Class, creates the application window and determines how the application window is shown.

See also *Create* method, *CreateForm* method, *CreateHandle* method, *Handle* property, *MainForm* property, *Run* method

Destroy

Destroy disposes of the application instance and all of its associated memory.

```
destructor Destroy; override;
```

Description Never call *Application.Destroy* or *Application.Free*. Use *Application.Terminate* to end an application.

Destroy signals that the application has been deactivated before calling the inherited *Destroy*. It then frees memory for hooked windows, the object instance, objects and lists that were created at application startup or that accumulated during execution

Since the Application (should) owns all forms, it will destroy all forms in its destructor.

See also *Terminate* method

HandleException

HandleException provides default handling of exceptions for the application.

```
procedure HandleException(Sender: TObject);
```

Description *HandleException* is useful for component writers when implementing custom components that generate an event that was not in response to a Windows message.

To assign other exception handling code for the application, use the *OnException* event handler.

If an exception passes through all the try blocks in the application code, the application automatically calls the *HandleException* method, which displays a dialog box indicating an error occurred. Unless the exception object is *EAbort*, *HandleException* calls the *OnException* handler, if one exists, otherwise it calls *ShowException* to display a message dialog box indicating an error occurred.

See also *EAbort* object, *OnException* event, *ShowException* method

HandleMessage

HandleMessage interrupts the execution of an application so that Windows can process a single message in the Windows message queue and then returns control to the application.

```
procedure HandleMessaage;
```

Description If the message queue is empty, *HandleMessage* calls the application's *Idle* method, which, in turn, calls the *OnIdle* event handler if one is defined.

Note If the application goes idle *HandleMessage* may take a long time to return. Therefore, do not call *HandleMessage* when waiting for something message-based, while priority actions are also being processed. In contrast, call *ProcessMessages* when processing other actions besides messages.

See also *OnIdle* event, *ProcessMessages* method

HelpCommand

HelpCommand provides quick access to any of the Help commands in the WinHelp API (application programming interface).

```
function HelpCommand(Command: Word; Data: Longint): Boolean;
```

Description Use *HelpCommand* to access WinHelp.

HelpCommand is a wrapper function that simply calls a method which calls the *OnHelp* event.

For information about these commands and the data passed to them, see the WinHelp topic in the Windows Help system.

See also *HelpContext* method, *HelpFile* property, *HelpJump* method, On*Help* event

HelpContext

HelpContext displays the Help file for an application.

```
type THelpContext = -MaxLongInt..MaxLongInt;

function HelpContext(Context: THelpContext): Boolean;
```

Description Use *HelpContext* to bring up the Help file for the application specified in the *HelpFile* property. *HelpContext* calls *OnHelp* directly.

HelpContext passes the file name contained in *HelpFile* and the context number passed in *Context* parameter. For example, if the *Context* value is 714, the *HelpContext* method displays the screen with the context help ID of 714 in the Help file.

HelpContext returns *False* if *HelpFile* is an empty string, meaning the application has no Help file assigned. In all other cases, *HelpContext* returns *True*.

See also *HelpCommand* method, *TWinControl.HelpContext* property, *HelpFile* property, *HelpJump* method, *OnHelp* event

HelpJump

HelpJump calls *OnHelp* indirectly.

```
function HelpJump(const JumpID: string): Boolean;
```

Description Use *HelpJump* to bring up Help for the specific topic. *HelpJump* passes the file name contained in *HelpFile* and the context string specified in the *JumpID*

parameter. For example, if the *JumpID* value is 'vclDefaultProperty', the *HelpJump* method displays the screen in the Help file that has the context string 'vclDefaultProperty'.

HelpJump returns *False* if *HelpFile* is an empty string, meaning the application has no Help file assigned. In all other cases, *HelpJump* returns *True*.

See also *HelpCommand* method, *HelpContext* method, *HelpFile* property, On*Help* event

HideHint

HideHint hides the current hint.

```
procedure HideHint;
```

Description Applications rarely, if ever, call *HideHint* directly. *HideHint* is used internally in Delphi to hide the hint window at the appropriate times according to predetermined conditions and timer values. For example, *CancelHint* calls *HideHint*.

See also *CancelHint* method, *lHint* property, *HintPause* property, *HintShortPause* property

HintMouseMessage

HintMouseMessage regulates the display of hint windows.

```
procedure HintMouseMessage(Control: TControl; var Message: TMessage);
```

Description Applications rarely, if ever, call *HintMouseMessage* directly. *HintMouseMessage* is used internally by Delphi to manage the display of the hint windows.

See also *CancelHint* method, *lHint* property, *HintPause* property, *HintShortPause* property

HookMainWindow

HookMainWindow enables a non-Delphi dialog box to receive messages sent to its parent, represented by the application object's *Handle* property.

```
type TWindowHook = function(var Message: TMessage): Boolean of object;

procedure HookMainWindow(Hook: TWindowHook);
```

Description Use *HookMainWindow* to ensure that the non-Delphi dialog box behaves correctly as a child of the application, not as a stand-alone window. For example, switching among applications with *Alt+Tab* treats the application as a single task after calling *HookMainWindow*, rather than treating the non-Delphi dialog box as a separate task.

The *TWindowHook* type is the parameter type used for the *HookMainWindow* method. It is a method-pointer type used for the dialog-procedures of non-Delphi dialog boxes. The dialog procedure is similar to a window procedure

for a window, in that it processes messages for the dialog box, but its syntax is somewhat different.

There is no problem with leaving a dialog box hooked into the main window, even for extended periods. However, should the dialog box close, call the *UnhookMainWindow* method to release the hook.

See also *Handle* property, *MainWindow* property, *UnhookMainWindow* method

Initialize

Initialize is the first method called for every Delphi project, providing an opportunity to call OLE automation, and other sub-systems, to be initialized.

```
procedure Initalize;
```

Description Use *Initialize* to call a procedure for OLE automation. *Initialize* automatically calls the *InitProc* procedure pointer, which is a pointer to a procedure that handles OLE automation.

Since *Initialize* is the first method called for a Delphi application, the *InitProc* pointer must somehow be predefined. By default the call to *Initialize* for the application does nothing, because it calls a default *InitProc* pointer that is **nil**.

Therefore, to call an OLE automation procedure either

- Add a unit that defines *InitProc*, such as the *ComObj* unit.
- Define a custom procedure for OLE automation, assign it to the *InitProc* pointer, and add this as the first line of code, that is, before the call to *Initialize*.

For projects that do not contain OLE automation servers, the call to *Initialize* can be safely deleted.

Note Although Initialize is the first method called in the main project source code, it is not the first code that is executed in an application. The application will first execute the "initialization" section of all the units used by the application.

MessageBox

MessageBox displays a specified message to the user.

```
function MessageBox(Text, Caption: PChar; Flags: Word): Integer;
```

Description Use *MessageBox* to displays a generic dialog box a message and one or more buttons. *Caption* is the caption of the dialog box and is optional.

MessageBox is an encapsulation of the Windows API *MessageBox* function. TApplication's encapsulation of *MessageBox* automatically supplies the missing window handle parameter needed for the Windows API function.

The value of the *Text* parameter is the message, which can be longer than 255 characters if necessary. Long messages are automatically wrapped in the

message box. The value of the *Caption* property is the caption that appears in the title bar of the dialog box. Captions can be longer than 255 characters, but they don't wrap. A long caption *results* in a wide message box.

To see the possible values of the *Flags* parameter, see the *MessageBox* function in the Windows API Help file. The corresponding parameter on that Help screen is called *TextType*. The values determine the buttons that appear in the message box and the behavior of the message box. The values can be combined to obtain the desired effect.

The return value of the *MessageBox* method is 0, if there wasn't enough memory to create the message box, or one of these values:

Value	Numeric value	Meaning
IDABORT	3	The user chose the Abort button.
IDCANCEL	2	The user chose the Cancel button.
IDIGNORE	5	The user chose the Ignore button.
IDNO	7	The user chose the No button.
IDOK	1	The user chose the OK button.
IDRETRY	4	The user chose the Retry button.
IDYES	6	The user chose the Yes button.

Minimize

Minimize shrinks an application to the Windows task bar.

```
procedure Minimize;
```

Description Call *Minimize* directly to minimize the application. When the user minimizes the application, the *Minimize* method is automatically called.

Note Don't confuse the *Minimize* method with minimizing a form or window. To minimize, maximize, or restore a window or form, change the value of the *WindowState* property.

See also *Icon* property, *OnMinimize* event, *Restore* method, *TForm.WindowState* property

NormalizeTopMosts

NormalizeTopMosts makes forms that have been designated as topmost forms (their *FormStyle* is *fsStayOnTop*) behave as if they were not topmost forms.

```
procedure NormalizeTopMosts;
```

Description Use *NormalizeTopMosts* to have a message box or dialog box that is displayed using the Windows API functions (such as MessageBox and MessageDlg) directly, appear on top of a topmost form. Otherwise the form remains on top, which may obscure the message box.

Calling *NormalizeTopMosts* is not necessary when using Delphi methods and functions to display message boxes.

To return the forms designated as *fsStayOnTop* to be topmost again, call *RestoreTopMosts*.

See also *TForm.FormStyle* property, *RestoreTopMosts* method

ProcessMessages

ProcessMessages interrupts the execution of an application so that Windows can respond to events.

```
procedure ProcessMessages;
```

Description Call *ProcessMessages* to permit Windows to process these events at the time *ProcessMessages* is called. The *ProcessMessages* method cycles the Windows message loop until it is empty and then returns control to the application.

Note In Win32, neglecting message processing only the application calling *ProcessMessages*, not other applications. In lengthy operations calling *ProcessMessages* periodically will allow the application to respond to paint and other messages.

Note *ProcessMessages* does not allow the application to go idle, whereas *HandleMessages* does.

See also *HandleMessages* method, *OnMessage* event

Restore

Restore restores a minimized application to its normal size.

```
procedure Restore;
```

Description Use *Restore* to restore the application to its previous size before it was minimized. When the user restores the application to normal size, the *Restore* method is automatically called.

Note Don't confuse the *Restore* method with restoring a form or window to its original size. To minimize, maximize, and restore a window or form, change the value of the *WindowState* property.

See also *TForm.WindowState* property

RestoreTopMosts

RestoreTopMosts restores forms designated as *fsStayOnTop* to be topmost again.

```
procedure RestoreTopMosts;
```

Description Use *RestoreTopMosts* to return forms that were originally designated as topmost forms (*FormStyle* is *fsStayOnTop*), then temporarily changed to be

non-topmost forms with the *NormalizeTopMosts* method, to a topmost position.

See also *NormalizeTopMosts* method

Run

Run executes the application.

```
procedure Run;
```

Description *Run* contains the application's main message loop. The call to *Run* does not return until the application's message loop is terminated.

When creating a new project, Delphi automatically creates a main program block in the project file that calls the *Run* method. When the application is executed, the application's *Run* method is called.

See also *Create* method, *CreateForm* method, *CreateHandle* method, *MainForm* property, *Terminate* method

ShowException

ShowException displays an exception that occurred in an application in a message box.

```
procedure ShowException(E: Exception);
```

Description *ShowException* is called by *HandleException* as a default if no handler is specified for the *OnException* event.

If an exception handler filters exceptions, *ShowException* will need to be called for those exceptions that are not filtered out.

To specify how exceptions are handled for an application write a handler for the *OnException* event.

Note Calling *ShowException* is rarely necessary in a Delphi application, since the default VCL exception handler will call *ShowException* automatically.

See also *HandleException* method, *OnException* event

Terminate

Terminate ends application execution.

```
procedure Terminate;
```

Description *Terminate* calls the Windows API *PostQuitMessage* function to perform an orderly shutdown of the application. *Terminate* is not immediate.

Terminate is called automatically on a WM_QUIT message and when the main form closes.

See also *Terminated* property

UnhookMainWindow

UnhookMainWindow releases the dialog procedure previously hooked by a call to the *HookMainWindow* method.

```
procedure UnhookMainWindow(Hook: TWindowHook);
```

Description Use *UnhookMainWindow* to release the hooked window. Specify the dialog procedure as the value of the *Hook* parameter.

The *TWindowHook* type is the parameter type used for the *InhookMainWindow* method. It is a method-pointer type used for the dialog-procedures of non-Delphi dialog boxes. The dialog procedure is similar to a window procedure for a window, in that it processes messages for the dialog box, but its syntax is somewhat different.

For more information about Windows hooks see the SetWindowsEx and UnhookWindowsEx functions in the Windows Help File.

See also *HookMainWindow* method

TAutoIncField

Unit: DB

TObject • TPersistent • TComponent • TField • TNumericField • TIntegerField • TAutoIncField

TAutoIncField represents an autoincrement field in a dataset.

Description Autoincrement fields can hold values in the range –2,147,483,648 to 2,147,483,647, and are used as a counter for the records in the dataset. Each record is given a unique value in an autoincrement field, with each record receiving the next highest integer from that of the previously inserted record. Autoincrement fields are commonly used to provide a unique primary key value for a record.

TAutoIncField does not introduce any new properties, events, or methods beyond those provided by the *TIntegerField* class. It exists as a separate class to represent the *DataType* for an autoincrement field.

See also *TDataSet* object

Properties

The following table lists the properties of *TAutoIncField*:

Property	Visibility	Default	Documented in
Alignment	published	taRightJustify	TField
AsBoolean	public		TField
AsCurrency	public		TField

Property	Visibility	Default	Documented in
AsDateTime	public		TField
AsFloat	public		TIntegerField
AsInteger	public		TIntegerField
AsString	public		TIntegerField
AsVariant	public		TIntegerField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
CurValue	public, read-only		TField
CustomConstraint	published		TField
DataSet	public		TField
DataSize	public, read-only		TIntegerField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayFormat	published		TNumericField
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditFormat	published		TNumericField
EditMask	public		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField

Property	Visibility	Default	Documented in
LookupList	public, read-only		TField
LookupResultField	published		TField
MaxValue	published	0	TIntegerField
MinValue	published	0	TIntegerField
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	public		TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TIntegerField
VCLComObject	public		TComponent
Visible	published	True	TField

Events

The following table lists the events of *TAutoIncField*:

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TAutoIncField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject

Method	Visibility	Documented in
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TAutoIncField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create creates an instance of *TAutoIncField*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TAutoIncField*. After calling the inherited constructor, *Create* sets the *DataType* property to *ftAutoInc*.

See also *DataType* property, *TIntegerField.Create* method

TBatchMove

Unit: DBTables | Palette: Data Access

TObject • TPersistent • TComponent • TBatchMove

TBatchMove enables applications to perform database operations on groups of records or entire tables.

Description Instantiate a *TBatchMove* object to

- Add the records in a dataset to a database table.
- Delete the records in a dataset from a database table.
- Copy a dataset to create a new database table or overwrite an existing table.

Set the properties of the *TBatchMove* object to specify the desired operation. Call the *Execute* method to perform the actual operation. *TTable* objects can also perform the same batch operations, but only with the options specified by the default *TBatchMove* property values.

TBatchMove has several properties that instruct the Borland Database Engine (BDE) how to handle problems that occur when executing the batch operation. Additionally, applications can use a *TBDECallback* object to further refine the response to problems that arise.

See also *TBDECallback* object, *TTable.BatchMove* method

Properties

The following table lists the properties of *TBatchMove*:

Property	Visibility	Default	Documented in
AbortOnKeyViol	published	True	TBatchMove
AbortOnProblem	published	True	TBatchMove
ChangedCount	public, read-only		TBatchMove
ChangedTableName	published		TBatchMove
CommitCount	published	0	TBatchMove
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent

Property	Visibility	Default	Documented in
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Destination	published		TBatchMove
KeyViolCount	public, read-only		TBatchMove
KeyViolTableName	published		TBatchMove
Mappings	published		TBatchMove
Mode	published	batAppend	TBatchMove
MovedCount	public, read-only		TBatchMove
Name	published		TComponent
Owner	public, read-only		TComponent
ProblemCount	public, read-only		TBatchMove
ProblemTableName	published		TBatchMove
RecordCount	published	0	TBatchMove
Source	published		TBatchMove
Tag	published	0	TComponent
Transliterate	published	True	TBatchMove
VCLComObject	public		TComponent

AbortOnKeyViol

AbortOnKeyViol specifies whether the batch operation is terminated immediately when integrity or key violations occur.

property AbortOnKeyViol: **Boolean**;

Description

Set *AbortOnKeyViol* to *True* to cancel the batch operation when it causes integrity or key violations in the destination table. Set *AbortOnKeyViol* to *False* to cause records that would cause integrity or key violations to be posted to a key violations table instead. Provide a *KeyViolTableName* when setting *AbortOnKeyViol* to *False*. If there is no *KeyViolTableName*, *AbortOnKeyViol* may still be set to *False*, in which case problem records are counted but not saved in a table.

Key violations occur when appending records to a table that has a primary key. On a table with a primary key, no two records may have the same values on all of the key fields. If a record that is being appended has the same values on all the key fields as a record in the destination table, it cannot be appended.

Integrity violations occur when deleting, appending or updating records to a table that has a referential integrity specification. A referential integrity specification requires that certain fields in one table must have values that exist in a particular field of another table. If a record that is being added does not match the restrictions imposed by referential integrity, it cannot be

added. If deleting record would cause tighter referential integrity restrictions that another table cannot meet, it cannot be deleted.

See also *KeyViolCount* property, *KeyViolTableName* property

AbortOnProblem

AbortOnProblem specifies whether the batch operation is terminated immediately when it is necessary to truncate data to make it fit in the specified *Destination*.

```
property AbortOnProblem: Boolean;
```

Description Set *AbortOnProblem* to *True* to cancel the batch operation when types of the fields in the *Source* table records do not match the types of the fields in the *Destination* table records. Set *AbortOnProblem* to *False* to cause the mismatched fields to be "trimmed". When *AbortOnProblem* is *False*, set the *ProblemTableName* to have all the untrimmed versions of records that were trimmed saved to a separate table.

Trimming a field means converting its value to a value compatible with the destination field type. For example, if a field in the destination table holds a string of size 10 characters, and the corresponding field in the source table holds a string of size 15 characters, any values from the source table longer than 10 characters must be truncated. Values that cannot be converted will cause *TBatchMove* to raise an exception.

Trimming does not occur when the *Mode* property is set to *batCopy*. Trimming fields on a delete operation means that records in the destination table may be deleted that do not exactly match the values from the source table.

Note To allow trimming on a field by field basis, use a *TBDECallback* object.

See also *Mode* property, *ProblemCount* property, *ProblemTableName* property, *TBDECallback* object

ChangedCount

ChangedCount indicates the number of records from the *Destination* table are altered as a result of the batch operation.

```
property ChangedCount: Integer;
```

Description Use *ChangedCount* to learn the number of records in the *Destination* table that are updated if the *Mode* is *batUpdate* or *batAppendUpdate*, or to learn the number that are deleted if the *Mode* is *batDelete*.

Set the *ChangedTableName* property to save the unchanged versions of the altered records to a separate table. *ChangedCount* is then the number of records in that separate table.

See also *ChangedTableName* property, *Destination* property, *Mode* property

ChangedTableName

ChangedTableName specifies the name of a local Paradox table that is created to hold copies of all records from the destination table that are changed by the batch operation.

property ChangedTableName: TFileName;

Description Set *ChangedTableName* to create a table that saves all unchanged versions of records that are changed by the batch operation. *ChangedTableName* must specify the name of a Paradox table. Setting *ChangedTableName* ensures that no information in the *Destination* table gets lost when the records are altered or deleted.

When the *Mode* property is *batUpdate* or *batAppendUpdate*, the new table will hold all records that were updated. When the *Mode* property is *batDelete*, the new table will hold all records that were deleted.

The *ChangedCount* property specifies the number of records in the new table.

See also *ChangedCount* property, *Destination* property, *Mode* property

CommitCount

CommitCount specifies how many records are batch moved before a commit occurs.

property CommitCount: Integer;

Description Use *CommitCount* to chunk a batch operations into smaller operations, *CommitCount* allows an application to accommodate server transaction logs that are not big enough to handle the entire batch. Setting *CommitCount* to 0 will cause chunking into the number of records that will fit in 32K bytes.

Note If the *Execute* method is called within an explicit transaction on the database containing the tables, *CommitCount* is ignored.

See also *Execute* method, *TDatabase.EndTransaction* method, *TDatabase.StartTransaction* method

Destination

Destination specifies the *TTable* object for the database table that is the destination of the batch operation.

property Destination: TTable;

Description Use *Destination* to indicate the database table that will be created or altered by the batch operation. The batch operation affects the *Destination* differently depending on the *Mode*:

Mode	Effect on *Destination*
batAppend	Records are added to the destination table. No pre-existing records are altered. Destination table must already exist.
batUpdate	Records in the destination table with key fields matching the corresponding fields in the source are changed to match the source records. No new records are added. Destination table must already exist and must have an index defined.
batAppendUpdate	Records in the destination table with key fields matching the corresponding fields in the source are changed to match the source. Source records that do not match key fields are added to the destination table. Destination table must already exist and must have an index defined.
batDelete	Records in the destination table that match the records in the source are deleted. Destination table must already exist and must have an index defined.
batCopy	Destination table is created to match the records in the source. If Destination already exists, it is replaced by the copy of the source.

See also *Mode* property, *Source* property

KeyViolCount

KeyViolCount reports the number of records which could not be replaced, added, or deleted from *Destination* because of integrity or key violations.

```
property KeyViolCount: Longint;
```

Description Use *KeyViolCount* to learn the number of records form the *Source* table that could not be applied to the *Destination* table because of integrity or key violations. To obtain a table of the offending records from the *Source* table, specify a *KeyViolTableName*.

Key violations occur when appending records to a table that has a primary key. On a table with a primary key, no two records may have the same values on all of the key fields. If a record that is being appended has the same values on all the key fields as a record in the destination table, it cannot be appended.

Integrity violations occur when deleting, appending or updating records to a table that has a referential integrity specification. A referential integrity specification requires that certain fields in one table must have values that exist in a particular field of another table. If a record that is being added does not match the restrictions imposed by referential integrity, it cannot be added. If deleting record would cause tighter referential integrity restrictions that another table cannot meet, it cannot be deleted.

Note If *AbortOnKeyViol* is *True*, *KeyViolCount* will never be greater than one, because the first violation will cause the batch operation to terminate.

See also *AbortOnKeyViol* property, *Destination* property, *KeyViolTableName* property, *Mode* property, *Source* property

KeyViolTableName

KeyViolTableName specifies the name of a Paradox table that will be created to contain all records from the *Source* that cannot participate in the batch operation because of integrity or key violations.


```
property KeyViolTableName: TTable;
```

Description Use *KeyViolTableName* to obtain a table of records that could not be applied to the *Destination* table because of integrity or key violations. *KeyViolCount* has the number of records in the table that is created.

Key violations occur when appending records to a table that has a primary key. On a table with a primary key, no two records may have the same values on all of the key fields. If a record that is being appended has the same values on all the key fields as a record in the destination table, it cannot be appended.

Integrity violations occur when deleting, appending or updating records to a table that has a referential integrity specification. A referential integrity specification requires that certain fields in one table must have values that exist in a particular field of another table. If a record that is being added does not match the restrictions imposed by referential integrity, it cannot be added. If deleting record would cause tighter referential integrity restrictions that another table cannot meet, it cannot be deleted.

Note If *AbortOnKeyViol* is *True*, then there will be at most one record in this table because the operation aborts with that first record.

See also *AbortOnKeyViol* property, *Destination* property, *KeyViolCount* property, *Source* Property

Mappings

Mappings specifies the column mappings for a batch operation.

```
property Mappings: TSTrings;
```

Description Set *Mappings* to specify the correspondence between fields in the *Source* and fields in the *Destination*. By default *TBatchMove* matches fields based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on. *Mappings* enables an application to override this default.

Mappings is a list of column mappings (one per line) in one of two forms. To map the column *ColName* in the source table to the column of the same name in the destination table, use

ColName

To map the column named *SourceColName* in the source table to the column named *DestColName* in the destination table, use

DestColName = SourceColName

When adding or appending records, fields in *Destination* which have no entry in *Mappings* will be set to NULL. When copying a dataset, fields in *Destination* which have no entry in *Mappings* will not appear as columns in the copy of the table.

If source and destination column data types are not the same, *TBatchMove* can either cancel the batch operation or perform a "best fit". If *AbortOnProblem* is *True*, type mismatches will terminate the batch move. Otherwise, the field values from the *Source* will be converted, if possible, to be compatible with the *Destination*.

See also *AbortOnProblem* property, *Destination* property, *Mode* property, *Source* property

Mode

Mode specifies what the *TBatchMove* object does when the *Execute* method is called.

property Mode: TBatchMode;

Description Use *Mode* to indicate whether the *TBatchMove* object should add records, replace records, delete records, or copy the *Source*. These are the possible values for *Mode*:

Value	Meaning
batAppend	Append the records in the source to the destination table. The destination table must already exist. This is the default mode.
batUpdate	Replace records in the destination table with matching records from the source table. The destination table must exist and must have an index defined to match records.
batAppendUpdate	If a matching record exists in the destination table, replace it. Otherwise, append records to the destination table. The destination table must exist and must have an index defined to match records.
batCopy	Create the destination table based on the structure of the source table. If the destination already exists, the operation will delete it, and replace it with the new copy of the source.
batDelete	Delete records in the destination table that match records in the source table. The destination table must already exist and must have an index defined.

See also *Destination* property, *Execute* method, *Source* property

MovedCount

MovedCount reports the number of records form the *Source* which were applied to the destination.

property MovedCount: Longint;

Description Read *MovedCount* to learn the number of records from the *Source* that were read during the *Execute* method. The *MovedCount* value includes any records which caused key or integrity violations, or which were trimmed.

See also *Execute* method, *Source* property

ProblemCount

ProblemCount indicates the number of records which could not be added to *Destination* without loss of data due to a field type mismatch.

property ProblemCount: Longint;

Description Read *ProblemCount* to *learn the number of records from the Source that had field values which could not be mapped to* destination fields without being "trimmed". If a *ProblemTableName* is specified, *ProblemCount* is the number of records in that table.

Trimming a field means converting its value to a value compatible with the destination field type. For example, if a field in the destination table holds a string of size 10 characters, and the corresponding field in the source table holds a string of size 15 characters, any values from the source table longer than 10 characters must be truncated. Values that cannot be converted will cause *TBatchMove* to raise an exception.

Trimming does not occur when the *Mode* property is set to *batCopy*. Trimming fields on a delete operation means that records in the destination table may be deleted that do not exactly match the values from the source table.

Note If *AbortOnProblem* is *True*, then *ProblemCount* will be at most one, because the operation aborts when the problem occurs.

See also *AbortOnProblem* property, *Destination* property, *Mode* property, *ProblemTableName* property, *Source* property

ProblemTableName

ProblemTableName specifies the name of a Paradox table that will be created to contain all records from the *Source* that contain fields which were trimmed to match the corresponding field type in the *Destination* table.

property ProblemTableName: TTable;

Description Set *ProblemTableName* to create a table that holds records from the *Source* that contain fields that had to be trimmed to be applied to the *Destination*.

ProblemCount will have the number of records placed in the new table. If *ProblemTableName* is not specified, the data in the record is still trimmed when applied to the destination table, but there will be no easy way to tell which fields were trimmed.

Trimming a field means converting its value to a value compatible with the destination field type. For example, if a field in the destination table holds a string of size 10 characters, and the corresponding field in the source table holds a string of size 15 characters, any values from the source table longer than 10 characters must be truncated. Values that cannot be converted will cause *TBatchMove* to raise an exception.

Trimming does not occur when the *Mode* property is set to *batCopy*. Trimming fields on a delete operation means that records in the destination table may be deleted that do not exactly match the values from the source table.

Note If *AbortOnProblem* is *True*, then there will be at most one record in this table because the operation aborts with the first record that has a problem.

See also *AbortOnProblem* property, *Destination* property, *Mode* property, *ProblemTableCount* property, *Source* property

RecordCount

RecordCount specifies the maximum number of records that are applied to the *Destination* when *Execute* is called.

```
property RecordCount: Longint;
```

Description Use *RecordCount* to limit the number of *Source records* that are applied to the *Destination*. If zero, all records are added, subtracted, or copied, beginning with the first record in *Source*. If *RecordCount* is greater than zero, up to *RecordCount* records are applied to the *Destination*, starting with the current record in the dataset. If *RecordCount* exceeds the number of records remaining in *Source*, no wraparound occurs; the operation is terminated.

See also *CommitCount* property, *Destination* property, *MovedCount* property, *Source* property

Source

Source specifies the *dataset* that is the source of the batch operation.

```
property Source: TBDEDataSet;
```

Description Use *Source* to indicate the dataset that contains the records that will be added to, subtracted from, or copied to the *Destination*. The *Mode* property indicates how the *Source* records are applied to the *Destination:*

Mode	Effect on *Destination*
batAppend	Records are added to the destination table.
batUpdate	Records in the destination table with key fields matching the corresponding fields in the source are changed to match the source records.
batAppendUpdate	Records in the destination table with key fields matching the corresponding fields in the source are changed to match the source. Source records that do not match key fields are added to the destination table.
batDelete	Records in the destination table that match the records in the source are deleted.
batCopy	Destination table is created to match the records in the source.

See also *Destination* property, *Mode* property

B

Transliterate

Transliterate specifies whether the data in the *Source* records should be converted form the locale of the *Source* to the locale of the *Destination* when the *Execute* method is called.

```
property Transliterate: Boolean;
```

Description Set *Transliterate* to *True* when the *Source* dataset and the *Destination* table use different language drivers and the data may contain extended ascii characters. Set *Transliterate* to *False* to avoid the overhead of the character set conversion when both the *Source* and the *Destination* use the same language driver or the data does not contain extended ascii characters.

See also *Destination* property, *Execute* method, *Source* property, *TDataSet.Locale* property

Methods

The following table lists the methods of *TBatchMove*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject

Method	Visibility	Documented in
Create	public	TBatchMove
DefaultHandler	public	TObject
Destroy	public	TBatchMove
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
Execute	public	TBatchMove
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Create

Create creates an instance of *TBatchMove*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create an instance of *TBatchMove* for performing a batch operation. *Create* allocates the memory for the *TBatchMove* object and

- Sets *AbortOnKeyViolation* to *True*.
- Sets *AbortOnProblem* to *True*.
- Sets *Transliterate* to *True*.
- Creates a *TStringList* object to hold the *Mappings*.

The property settings made by *Create* are the values used when a *TTable* object performs batch operations using its *BatchMove* method.

See also *AbortOnKeyViolation* property, *AbortOnProblem* property, *Mappings* property, *Transliterate* property, *TTable.BatchMove* method

Destroy

Destroy destroys an instance of *TBatchMove*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TBatchMove* object is not already freed and only then calls *Destroy*.

Before calling the inherited destructor, *Destroy* frees the *TStringList* object that implements the *Mappings* property.

See also *Free* method, *Mappings* property

Execute

Execute performs the batch operation specified by *Mode*.

procedure Execute;

Description After setting properties to indicate what batch operation should be performed and how, call *Execute* to perform the operation. As a minimum, the *Mode*, *Source*, and *Destination* properties must be set.

After calling *Execute*, the values of the *ChangedCount*, *KeyViolCount*, *MovedCount*, *ProblemCount*, and *RecordCount* properties will become available. These properties will give an indication of what happened as a result of the call to *Execute*. Additional information about the batch operation will be available through the creation of a changed table, a keyviol table, and/or a problems table if it was requested.

See also *ChangedCount* property, *ChangedTableName* property, *Destination* property, *KeyViolCount* property, *KeyViolTableName* property, *Mode* property, *MovedCount* property, *ProblemCount* property, *ProblemTableName* property, *RecordCount* property, *Source* property

TBCDField

Unit: DB

TObject •TPersistent •TComponent •TField •TNumericField •TBCDField

TBCDField represents a binary-coded decimal (BCD) field in a dataset.

Description BCD values provide greater precision and accuracy than floating-point numbers. BCD fields are often used for storing and manipulating monetary values. Normally, BCD values are stored and manipulated in decimal (base

ten). However, Delphi does not have a native binary-coded decimal type, so *TBCDField* uses the *Currency* type to manipulate BCD values. *Currency* is a binary (base two) format.

Because *TBCDField* uses the *Currency* type to work with data from BCD fields, *TBCDField* limits the precision of the BCD values it can support to 4 decimal places and 20 significant digits. The underlying database table may include values that require greater precision. When it encounters such values, *TBCDField* raises an exception.

As a descendant of *TNumericField*, *TBCDField* includes many properties, methods, and events that are useful for managing the value and properties of a numeric field in a database.

See also *TDataSet object*

Properties

The following table lists the properties of *TBCDField*:

Property	Visibility	Default	Documented in
Alignment	published	taRightJustify	TField
AsBoolean	public		TField
AsCurrency	public		TBCDField
AsDateTime	public		TField
AsFloat	public		TBCDField
AsInteger	public		TBCDField
AsString	public		TBCDField
AsVariant	public		TBCDField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
Currency	published	False	TBCDField
CurValue	public, read-only		TField
CustomConstraint	published		TField
DataSet	public		TField
DataSize	public, read-only		TBCDField
DataType	public, read-only		TField
DefaultExpression	published		TField

Property	Visibility	Default	Documented in
DesignInfo	public		TComponent
DisplayFormat	published		TNumericField
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditFormat	published		TNumericField
EditMask	public		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField
MaxValue	published		TBCDField
MinValue	published		TBCDField
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	published	4	TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TBCDField
VCLComObject	public		TComponent
Visible	published	True	TField

AsCurrency

AsCurrency converts the value of the field to or from a currency value.

property AsCurrency: Currency;

Description Use *AsCurrency* to

- Assign the binary-coded decimal field value to a currency value.
- Assign a currency value to the binary-coded decimal field.

For example, this code assigns a currency value to a binary-coded decimal field value:

```
MyBCDField.AsCurrency := CurrencyValue;
```

To assign a binary-coded decimal field value to a currency value, use code similar to this:

```
CurrencyVariable := MyBCDField.AsCurrency;
```

For BCD fields, the *AsCurrency* property is equivalent to the *Value* property. Use *AsCurrency* when you do not know whether or not the field component is a BCD field.

If a value assigned to the field is not within the range established by the *MinValue* and the *MaxValue* properties, an *EDatabaseError* exception is raised.

Note Don't confuse the *AsCurrency* property, which converts the type of the data, with the *Currency* property, which indicates how the data should be formatted.

See also *Currency* property, *MaxValue* property, *MinValue* property, *Value* property

AsFloat

AsFloat converts the value of the field to or from a floating-point value.

property AsFloat: Double;

Description Use *AsFloat* to

- Assign the binary-coded decimal field value to a floating-point value.
- Assign a floating-point value to the binary-coded decimal field.

For example, this code assigns a floating-point value to a binary-coded decimal field value:

```
MyBCDField.AsFloat := FloatVariable;
```

To assign a binary-coded decimal field value to a floating-point value, use code similar to this:

```
FloatVariable := MyBCDField.AsFloat;
```

If a value assigned to the field is not within the range established by the *MinValue* and the *MaxValue* properties, an *EDatabaseError* exception is raised.

See also *MaxValue* property, *MinValue* property

AsInteger

AsInteger converts the value of the field to or from an integer.

property AsInteger: Longint;

Description Use *AsInteger* to

- Assign the binary-coded decimal field value to an integer variable.
- Assign an integer value to the binary-coded decimal field.

For example, this code assigns integer value to a binary-coded decimal field value:

```
MyBCDField.AsInteger := IntegerVariable;
```

To assign a binary-coded decimal field value to an integer value, use code similar to this:

```
IntegerVariable := MyBCDField.AsInteger;
```

If a value assigned to the field is not within the range established by the *MinValue* and the *MaxValue* properties, an *EDatabaseError* exception is raised.

See also *MaxValue* property, *MinValue* property

AsString

AsString converts the value of the field to or from a binary-coded decimal value.

property AsString: **string**;

Description Use *AsString* to

- Assign the binary-coded decimal field value to string variable.
- Assign a string value to the binary-coded decimal field.

For example, this code assigns a string value to a binary-coded decimal field value:

```
MyBCDField.AsString := '12.32';
```

To assign a binary-coded decimal field value to a string value, use code similar to this:

```
NewString := MyBCDField.AsString;
```

If the string assigned to the field does not represent a valid currency value within the range specified by the *MinValue* and *MaxValue* properties, an *EDatabaseError* exception is raised.

See also *MaxValue* property, *MinValue* property

AsVariant

AsVariant converts the value of a binary-coded decimal field to or from a variant value.

property AsVariant: Variant;

Description Use *AsVariant* to

- Assign the binary-coded decimal field value to variant value.
- Assign a variant value to the binary-coded decimal field.

For example, this code assigns a Variant value to a binary-coded decimal field value:

```
MyBCDField.AsVariant := VariantVariable;
```

To assign a binary-coded decimal field value to a variant value, use code similar to this:

```
VariantVariable:= MyBCDField.AsVariant;
```

Using variants is useful when both the actual field type and the type of the data are unknown.

If a value assigned to the field is not within the range established by the *MinValue* and the *MaxValue* properties, an *EDatabaseError* exception is raised.

See also *AsCurrency* property, *AsFloat* property, *AsInteger* property, *AsString* property, *MaxValue* property, *MinValue* property

Currency

Currency determines whether the value in the field should be formatted as a currency value.

property Currency: Boolean;

Description Use the *Currency* property to influence how the value in the field should be formatted for viewing and editing in a data-aware control.

For display purposes, the field is formatted using the *DisplayFormat* property if it is assigned. If *DisplayFormat* is not assigned, *Currency* determines how the field is formatted for display. If *Currency* is *True*, the value is formatted for display using the *FloatToText* function with the *ffCurrency* formatting code. If *Currency* is *False*, the value is formatted with the *ffGeneral* format.

For editing purposes, the field is formatted using the *EditFormat* (or *DisplayFormat*) property if it is assigned. If neither *EditFormat* nor *DisplayFormat* is assigned, *Currency* determines how to format the field for editing. If *Currency* is *True*, the value is formatted for editing using the *FloatToText* function with the *ffFixed* formatting code. If *Currency* is *False*, the value is formatted with the *ffGeneral* format.

See also *DisplayFormat* property, *DisplayText* property, *EditFormat* property, *FloatToText* function, *FloatToTextFmt* function, *Text* property

DataSize

DataSize is the amount of memory that is needed to store the value of a BCD field in its native binary-coded decimal format.

property DataSize: Word;

Description Check *DataSize* to determine the number of bytes required to store a BCD field component's value. Use *DataSize* to determine the size needed for a buffer when working with the *GetData* and *SetData* methods. *GetData* and *SetData* work with the field's value in native database format rather than the *Currency* type used by the BCD field object.

For BCD fields, *DataSize* is always 34. BCD fields are a native type to Paradox tables that take up 34 bytes in a database record.

See also *GetData* method, *SetData* method, *Size* property

B

MaxValue

MaxValue limits the maximum value in the binary-coded decimal field.

property MaxValue: Currency;

Description Use *MaxValue* to get or set the maximum value that can be entered into the field. If a value greater than *MaxValue* is entered, an *EDatabaseError* exception is raised.

Newly created BCD fields set MaxValue to the maximum value that can be expressed by the *Currency* data type.

See also *MinValue* property

MinValue

MinValue limits the minimum value in the binary-coded decimal field.

property MaxValue: Currency;

Description Use *MinValue* to get or set the minimum value that can be entered into the field. If a value smaller than *MinValue* is entered, an *EDatabaseError* exception is raised.

Newly created integer fields set MinValue to the minimum value that can be expressed by the *Currency* data type.

See also *MaxValue* property

Value

Value is the value of the BCD field as a *Currency* value.

property Value: Currency;

Description Unlike other field types, the *Value* of a BCD field does not correspond exactly to the physical format of the data that is stored in the underlying database table. Delphi does not have a native type for binary-coded decimal. Therefore, *TBCDField* converts the data from a binary-coded decimal value to a *Currency* value when it fetches the data from the database table, and converts it from a *Currency* value to a binary-coded decimal value when it posts the data to the database. For BCD fields, *Value* is the same as the *AsCurrency* property.

Note Because *TBCDField* works with its data as a *Currency* value, simply getting the value of the field and posting that value right back to the database table can actually alter the contents of the field. Thus the line

```
MyBCDField.SetData(MyBCDField.Value);
```

can alter the contents of the field in the database table.

See also *AsCurrency* property, *SetData* method

Events

The following table lists the events of *TBCDField*:

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TBCDField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TBCDField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent

Method	Visibility	Documented in
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create creates an instance of *TBCDField*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TBCDField*. After calling the inherited constructor, *Create* sets the *DataType* to *ftBCD*, and initializes the *Size* to 4.

It is seldom necessary to explicitly call *Create* because a BCD field component is instantiated automatically for all BCD fields in a dataset.

See also *DataSet* property, *DataType* property, *Size* property, *TNumericField.Create* method

TBDEDataSet

Unit: DBTables

TObject • TPersistent • TComponent • TDataSet • TBDEDataSet

Encapsulates Borland Database Engine (BDE) functionality for descendent dataset objects.

Description *TBDEDataSet* is a dataset object that defines BDE functionality for a dataset. Applications never use *TBDEDataSet* objects directly. Instead they use the descendants of *TBDEDataSet* and *TDBDataSet*, such as *TQuery*, *TStoredProc*, and *TTable*, which inherit its database-related properties and methods.

Developers who create custom dataset components that work through the BDE may want to derive them directly from *TBDEDataSet* to inherit all the functionality of *TDataSet* and the BDE-related properties and methods of *TBDEDataSet*. To inherit control over Delphi database objects, however, most developers should derive custom dataset components that use the BDE from the immediate descendant of *TBDEDataSet*, *TDBDataSet*.

See also *TDataSet* object, *TDBDataSet* object, *TQuery* object, *TStoredProc* object, *TTable* object

Properties

The following table lists the properties of *TBDEDataSet*:

Property	Visibility	Default	Documented in
Active	published	False	TDataSet
AutoCalcFields	published	True	TDataSet
BOF	public, read-only		TDataSet
Bookmark	public		TDataSet
CacheBlobs	public	True	TBDEDataSet
CachedUpdates	published	False	TBDEDataSet
CanModify	public, read-only		TBDEDataSet
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DataSource	public, read-only		TDataSet

Property	Visibility	Default	Documented in
DefaultFields	public, read-only		TDataSet
Designer	public, read-only		TDataSet
DesignInfo	public		TComponent
EOF	public, read-only		TDataSet
ExpIndex	public, read-only		TBDEDataSet
FieldCount	public, read-only		TDataSet
FieldDefs	public		TDataSet
Fields	public		TDataSet
FieldValues	public		TDataSet
Filter	published		TBDEDataSet
Filtered	published	False	TBDEDataSet
FilterOptions	published	[]	TBDEDataSet
Found	public, read-only		TDataSet
Handle	public, read-only		TBDEDataSet
KeySize	public, read-only		TBDEDataSet
Locale	public, read-only		TBDEDataSet
Modified	public, read-only		TDataSet
Name	published		TDataSet
Owner	public, read-only		TComponent
RecNo	public		TBDEDataSet
RecordCount	public, read-only		TBDEDataSet
RecordSize	public, read-only		TBDEDataSet
State	public, read-only		TDataSet
Tag	published	0	TComponent
UpdateObject	public		TBDEDataSet
UpdateRecordTypes	public		TBDEDataSet
UpdatesPending	public, read-only		TBDEDataSet
VCLComObject	public		TComponent

B

CacheBlobs

Determines whether BLOBs are cached in memory.

property CacheBlobs: Boolean;

Description Use *CacheBlobs* to specify whether or not to store BLOB images in memory to improve performance when scrolling through records that display BLOB images. *CacheBlobs* is *True* by default, meaning that BLOBs are cached in memory. If an application does not need to display the BLOBs associated with records, then set *CacheBlobs* to *False* to conserve system resources.

CachedUpdates

Specifies whether cached updates are enabled for a dataset.

property CachedUpdates: Boolean;

Description *CachedUpdates* enables or disables the use of cached updates for a dataset. If *CachedUpdates* is *True*, cached updates are enabled. If *CachedUpdates* is *False*, cached updates are disabled.

When cached updates are enabled, updates to a dataset (such as posting changes, inserting new records, or deleting records), are stored in an internal cache on the client machine instead of being written directly to the dataset's underlying database tables. When changes are complete, an application writes all cached changes to the database in the context of a single transaction.

Cached updates are most useful to client applications working with remote database servers. The main benefits of enabling cached updates are

- Fewer transactions and shorter transaction times.
- Minimization of network traffic.

The potential drawbacks of enabling cached updates are

- Other applications can access and change the actual data on the server while users are editing local copies of the data, resulting in an update conflict when cached updates are applied to the database.
- Other applications cannot access data changes made by an application until its cached updates are applied to the database.

See also *UpdateObject* property, *UpdateRecordTypes* property, *UpdatesPending* property, *OnUpdateError* event, *OnUpdateRecord* event, *ApplyUpdates* method, *CancelUpdates* method, *CommitUpdates* method, *RevertRecord* method, *UpdateRecord* method, *TDatabase.ApplyUpdates* method, *TDatabase.Commit* method, *TDatabase.Rollback* method

CanModify

Indicates whether the database underlying a dataset permits write access to data.

property CanModify: Boolean;

Description Examine *CanModify* to determine if the database underlying a dataset permits write access to data. When a database connection is established, a dataset typically requests write access. *CanModify* indicates whether or not write access is granted. If *CanModify* is *True*, data can be modified and written to the database server. If *CanModify* is *False*, data can be viewed, but not modified.

Note Restrictions on write access for a given user may still prevent writing to an SQL database server even if *CanModify* is *True*.

See also *DataSource* property, *TDataSource.AutoEdit* property

ExpIndex

Indicates whether a dataset is using a dBASE expression index.

property ExpIndex: Boolean;

Description Read *ExpIndex* to determine if a dataset is using a dBASE expression index. If *ExpIndex* is *True*, an expression index is in use; otherwise *ExpIndex* is *False*.

Filter

Specifies the text of the current filter for a dataset.

property Filter: **string**;

Description Use *Filter* to specify a dataset filter. When filtering is applied to a dataset, only those records that meet a filter's conditions are available to an application. *Filter* contains the string that describes the filter condition. For example, the following filter condition displays only those records where the State field is 'CA' or 'MA':

```
State = 'CA' or State = 'MA'
```

To filter strings bases on partial comparisons, use an asterisk as a wildcard. For example,

```
State = 'M*'
```

Note Applications can set *Filter* at runtime to change the filtering condition for a dataset at (for example, in response to user input).

See also *Filtered* property, *FilterOptions* property, *FindFirst* method, *FindLast* method, *FindNext* method, *FindPrior* method, *OnFilterRecord* event

Filtered

Specifies whether filtering is active for a dataset.

property Filtered: Boolean;

Description Check *Filtered* to determine whether or not dataset filtering is in effect. If *Filtered* is *True*, then filtering is active. Otherwise *Filtered* is *False*. To apply filter conditions specified in the *Filter* property or the *OnFilterRecord* event handler, set *Filtered* to *True*.

Note When filtering is enabled, user edits to a record may mean that the record no longer meets a filter's test condition. The next time the record is retrieved from the dataset while the filter is in effect, the record may seem to disappear. If that happens, the next record that passes the filter condition becomes the current record.

See also *Filter* property, *FilterOptions* property, *FindFirst* method, *FindLast* method, *FindNext* method, *FindPrior* method, *OnFilterRecord* event,

FilterOptions

Specifies whether filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

```
type
  TFilterOption = (foCaseInsensitive, foNoPartialCompare);
  TFilterOptions = set of TFilterOption;

property FilterOptions: TFilterOptions
```

Description Set *FilterOptions* to specify whether or not filtering is case insensitive when filtering on string or character fields, and whether or not partial comparisons for matching filter conditions is allowed.

By default, *FilterOptions* is set to an empty set. For filters based on string fields, set *FilterOptions* to *foCaseInsensitive* to catch all variations on a string regardless of capitalization.

For filter conditions based on multiple conditions or fields, set *FilterOptions* to *foNoPartialCompare* to force exact matches only on comparison.

To prevent partial string comparisons, include *foNoPartialCompare* in *FilterOptions*.

Note To filter strings bases on partial comparisons, exclude *foNoPartialCompare* from *FilterOptions* and use an asterisk as a wildcard. For example,

```
State = 'M*'
```

See also *Filter* property, *Filtered* property, *OnFilterRecord* event, *Locate* method, *Lookup* method

Handle

Specifies the Borland Database Engine (BDE) cursor handle for the dataset.

```
type
  HDBICur: Longint;

property Handle: HDBICur;
```

Description Use *Handle* only to bypass *TDataSet* methods and call directly into the BDE. Many BDE function calls require a cursor handle parameter. *Handle* is assigned an initial value when a dataset is opened. If used with a BDE call that changes the current record position, call *Resync* immediately after returning from the BDE call.

Note Do not use this property unless an application requires BDE functionality not available through standard Delphi components.

See also *Locale* property, *Resync* method

KeySize

Specifies the size of the key for the current index of the dataset.

```
property KeySize: Word;
```

Description Check *KeySize* to determine the size, in bytes, of the key for the current index. *KeySize* varies depending on the number and type of fields that make up the current index.

Locale

Identifies the Borland Database Engine (BDE) language driver for the dataset.

```
type
  TLocale: Pointer;

property Locale: TLocale;
```

Description Examine *Locale* to determine the BDE language driver for the dataset. Applications that make direct calls to the BDE API may need to pass *Locale* information as a function parameter.

Language driver information may also be available as one of the values in the *Params* property of the database component that owns the dataset.

Note Do not use this property unless an application requires BDE functionality not available through standard Delphi components.

See also *Handle* property, *TDatabase.Locale* property

RecNo

Indicates the current record in the dataset.

```
property RecNo: Longint;
```

Description Examine *RecNo* to determine the record number of the current record in the dataset. Applications might use this property with *RecordCount* to iterate through all the records in a dataset, though typically record iteration is handled with calls to *First*, *Last*, *MoveBy*, *Next*, and *Prior*.

Note If accessing Paradox tables, *RecNo* can be set to a specific record number to position the cursor on that record.

See also *RecordCount* property, *RecordSize* property, *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

RecordCount

Indicates the total number of records associated with the dataset.

```
property RecordCount: Longint;
```

Description Examine *RecordCount* to determine the total number of records in the dataset. Applications might use this property with *RecNo* to iterate through all the records in a dataset, though typically record iteration is handled with calls to *First*, *Last*, *MoveBy*, and *Prior*.

Note Use *RecordCount* with care, because record counting can be a costly operation, especially for SQL queries that return large result sets. Generally, an application should only use *RecordCount* with Paradox and dBASE tables.

See also *RecNo* property, *RecordSize* property, *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

RecordSize

Indicates the size of a record in the dataset.

```
property RecordSize: Word;
```

Description Examine *RecordSize* to determine the physical size, in bytes, of the buffer Delphi allocates to hold a record in the dataset. When a dataset is opened, the *Open* procedure requests record-buffer size information from the Borland Database Engine (BDE) and stores the returned information in *RecordSize*. Delphi uses this information internally. Applications seldom, if ever, require this information.

See also *RecordCount* property, *RecNo* property, *Open* method

UpdateObject

Specifies the update object component used to update a read-only result set when cached updates are enabled.

```
type
  TDataSetUpdateObject = class(TComponent)

property UpdateObject: TDataSetUpdateObject;
```

Description Use *UpdateObject* to specify the *TUpdateObject* component to use in an application that must be able to update a read-only result set returned by the Borland Database Engine (BDE).

The BDE always attempts to provide an updatable, or "live" query result unless an application specifically requests a read-only view of data. In some cases, such as a query made against multiple tables, a live result set cannot be returned. In these cases, *UpdateObject* can be used to specify a *TUpdateSQL* component that performs updates as a separate transaction that is transparent to the application.

See also *CachedUpdates* property, *UpdateRecordTypes* property, *OnUpdateError* event, *OnUpdateRecord* event, *ApplyUpdates* method, *CommitUpdates* method, *RevertRecord* method, *TUpdateSQL* object, *UpdateRecord* method, *UpdateStatus* method

UpdateRecordTypes

Specifies the type of records visible in a dataset when cached updates are enabled.

```
type TUpdateRecordTypes = set of (rtModified, rtInserted, rtDeleted, rtUnmodified);

property UpdateRecordTypes: TUpdateRecordTypes;
```

Description Use *UpdateRecordTypes* to specify the records that are visible in a dataset when cached updates are enabled. *UpdateRecordTypes* is a set that can have the following values:

Value	Meaning
rtModified	Modified records are visible.
rtInserted	Inserted records are visible.
rtDeleted	Deleted records are visible.
rtUnmodified	Unmodified records are visible.

By default, a dataset is created with an *UpdateRecordTypes* set of [*rtModified, rtInserted, rtUnmodified*], meaning that all existing, edited, or inserted records are visible to the user.

An application that must cycle through a dataset to undelete records may change *UpdateRecordTypes* as part of an undelete method, so that deleted records are "visible" long enough to restore them to their previously undeleted conditions.

An application might also use *UpdateRecordTypes* like a filter to temporarily limit visible records to those added or inserted by the user during the current session.

See also *CachedUpdates* property, *UpdateObject* property, *OnUpdateError* event, *OnUpdateRecord* event, *RevertRecord* method, *UpdateStatus* method

UpdatesPending

Indicates whether the cached updates buffer contains records that are not yet applied.

```
property UpdatesPending: Boolean;
```

Description Examine *UpdatesPending* to check the status of the cached updates buffer. If *UpdatesPending* is *True*, then there are edited, deleted, or inserted records to apply to the database. If *UpdatesPending* is *False*, there are no records in the cache.

See also *CachedUpdates* property, *OnUpdateRecord* event, *UpdateStatus* method

Events

The following table lists the events of *TBDEDataSet*:

Event	Visibility	Documented in
AfterCancel	published	TDataSet
AfterClose	published	TDataSet
AfterDelete	published	TDataSet
AfterEdit	published	TDataSet
AfterInsert	published	TDataSet
AfterOpen	published	TDataSet
AfterPost	published	TDataSet
AfterScroll	published	TDataSet
BeforeCancel	published	TDataSet
BeforeClose	published	TDataSet
BeforeDelete	published	TDataSet
BeforeEdit	published	TDataSet
BeforeInsert	published	TDataSet
BeforeOpen	published	TDataSet
BeforePost	published	TDataSet
BeforeScroll	published	TDataSet
OnCalcFields	published	TDataSet
OnDeleteError	published	TDataSet
OnEditError	published	TDataSet
OnFilterRecord	published	TDataSet
OnNewRecord	published	TDataSet
OnPostError	published	TDataSet
OnServerYield	published	TBDEDataSet
OnUpdateError	published	TBDEDataSet
OnUpdateRecord	published	TBDEDataSet

OnServerYield

OnServerYield occurs during processing of a query on a remote database server when the server sends a yield call back to the Borland Database Engine (BDE).

```
type
  TOnServerYieldEvent = procedure(DataSet: TDataSet; var AbortQuery: Boolean) of object;

property OnServerYield: TOnServerYieldEvent;
```

Description Write an *OnServerYield* event handler to respond to a BDE callback generated when a database server yields temporary control to the BDE during a query

operation. When it receives such control, it calls *OnServerYield* if the handler exists.

To continue processing the query, *OnServerYield* should set the *AbortQuery* parameter to *False*. To end query processing set *AbortQuery* to *True*.

Note Currently, only Sybase yields control during query processing.

See also *TQuery* object

OnUpdateError

OnUpdateError occurs if an exception is generated when cached updates are applied to a database.

```
type
  TUpdateKind = (ukModify, ukInsert, ukDelete);
  TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);
  TUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;

property OnUpdateError: TUpdateErrorEvent;
```

Description Write an *OnUpdateError* event handler to respond to exceptions generated when cached updates are applied to a database.

Because there is a delay between the time a record is first cached and the time cached updates are applied, there is a possibility that another application may change one or more of the same records in the database before the cached changes can be applied. The Borland Database Engine (BDE) checks for this condition and raises an exception. Delphi responds by calling the *OnUpdateError* event handler if it exists.

DataSet is the name of the dataset to which updates are applied.

E is a pointer to a *EDBEngineError* object from which an application can extract an error message and the actual cause of the error condition. An *OnUpdateError* handler can use this information to determine how to respond to the error condition.

UpdateKind describes the type of update that generated the error. The following table lists the possible values for *UpdateKind* and what they indicate:

Value	Meaning
ukModify	Editing an existing record caused an error.
ukInsert	Inserting or appending a new record caused an error.
ukDelete	Deleting an existing record raised an error.

UpdateAction indicates the action to take when the *OnUpdateError* handler exits. On entry into the handler, *UpdateAction* is always set to *uaFail*. If *OnUpdateError* can handle or correct the error, set *UpdateAction* to *uaRetry*

before exiting the error handler. The following table lists the possible values for *UpdateAction* and what they indicate:

Value	Meaning
uaAbort	Aborts the update operation without displaying an error message.
uaApplied	Not used in error handling routines.
uaFail	Aborts the update operation and displays an error message.
uaRetry	Repeats the update operation that originally raised the error condition.
uaSkip	Skips updating the record that raised the error condition, and leaves the unapplied changes in the cache.

The error handler can use the *TField.OldValue* and *TField.NewValue* properties to evaluate error conditions and set *TField.NewValue* to a new value to reapply. In this case, set *UpdateAction* to *uaRetry* before exiting.

Note If a call to *ApplyUpdates* raises an exception and *ApplyUpdates* is not called within the context of a **try...except** block, an error message is displayed. If an *OnUpdateError* handler cannot correct the error condition and leaves *UpdateAction* set to *uaFail*, the error message is displayed twice. To prevent redisplay, set *UpdateAction* to *uaAbort* in the error handler.

Important The code in an *OnUpdateError* handler must not call any methods that make a different record the current one.

See also *ApplyUpdates* method, *CachedUpdates* property, *EDBEngineError* object, *OnUpdateRecord* event, *TField* object, *UpdateObject* property, *UpdateRecordTypes* property, *UpdatesPending* property,

OnUpdateRecord

OnUpdateRecord occurs when cached updates are applied to a record.

```
type
  TUpdateKind = (ukModify, ukInsert, ukDelete);
  TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);
  TUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind;
    var UpdateAction: TUpdateAction) of object;

property OnUpdateRecord: TUpdateRecordEvent;
```

Description Write an *OnUpdateRecord* event handler to process updates that cannot be handled by a single update component, such as implementation of cascading updates, insertions, or deletions. This handler is also useful for applications that require additional control over parameter substitution in update components.

DataSet is the name of the dataset to which updates are applied.

UpdateKind describes the type of update to perform. The following table lists the possible values for *UpdateKind* and what they indicate:

Value	Meaning
ukModify	Edit an existing record.
ukInsert	Insert or append a new record.
ukDelete	Delete an existing record.

UpdateAction indicates the action taken by the *OnUpdateRecord* handler before it exits. On entry into the handler, *UpdateAction* is always set to *uaFail*. If *OnUpdateRecord* is successful, it should set *UpdateAction* to *uaApplied* before exiting. The following table lists the possible values for *UpdateAction* and what they indicate:

Value	Meaning
uaAbort	Abort the update operation without displaying an error message.
uaApplied	Update is applied. Free update record from cache.
uaFail	Aborts the update operation and displays an error message.
uaRetry	Not used for record updates.
uaSkip	Update is skipped. Leave update record in the cache.

Note The code in an *OnUpdateRecord* handler must not call any methods that make a different record the current one.

See also *CachedUpdates* property, *UpdateObject* property, *UpdateRecordTypes* property, *UpdatesPending* property, *OnUpdateError* event, *ApplyUpdates* method, *CancelUpdates* method, *CommitUpdates* method, *RevertRecord* method, *UpdateRecord* method, *UpdateStatus* method

Methods

The following table lists the methods of *TBDEDataSet*:

Method	Visibility	Documented in
ActiveBuffer	public	TDataSet
Append	public	TDataSet
AppendRecord	public	TDataSet
ApplyUpdates	public	TBDEDataSet
Assign	public	TPersistent
BookmarkValid	public	TBDEDataSet
Cancel	public	TBDEDataSet
CancelUpdates	public	TBDEDataSet
CheckBrowseMode	public	TDataSet
ClassInfo	public	TObject
ClassName	public	TObject

Method	Visibility	Documented in
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClearFields	public	TDataSet
Close	public	TDataSet
CommitUpdates	public	TBDEDataSet
CompareBookmarks	public	TBDEDataSet
ControlsDisabled	public	TDataSet
Create	public	TBDEDataSet
CursorPosChanged	public	TDataSet
DefaultHandler	public	TObject
Delete	public	TDataSet
Destroy	public	TBDEDataSet
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableConstraints	public	TBDEDataSet
DisableControls	public	TDataSet
Dispatch	public	TObject
Edit	public	TDataSet
EnableConstraints	public	TBDEDataSet
EnableControls	public	TDataSet
FetchAll	public	TBDEDataSet
FieldAddress	public	TObject
FieldByName	public	TDataSet
FindComponent	public	TComponent
FindField	public	TDataSet
FindFirst	public	TDataSet
FindLast	public	TDataSet
FindNext	public	TDataSet
FindPrior	public	TDataSet
First	public	TDataSet
Free	public	TObject
FreeBookmark	public	TDataSet
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetBookmark	public	TDataSet
GetCurrentRecord	public	TBDEDataSet
GetFieldList	public	TDataSet
GetFieldNames	public	TDataSet
GetIndexInfo	public	TBDEDataSet

Method	Visibility	Documented in
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GotoBookmark	public	TDataSet
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
Insert	public	TDataSet
InsertComponent	public	TComponent
InsertRecord	public	TDataSet
InstanceSize	public	TObject
IsEmpty	public	TDataSet
IsLinkedTo	public	TDataSet
Last	public	TDataSet
Locate	public	TBDEDataSet
Lookup	public	TBDEDataSet
MethodAddress	public	TObject
MethodName	public	TObject
MoveBy	public	TDataSet
NewInstance	public	TObject
Next	public	TDataSet
Open	public	TDataSet
Post	public	TDataSet
Prior	public	TDataSet
Refresh	public	TDataSet
RemoveComponent	public	TComponent
Resync	public	TDataSet
RevertRecord	public	TBDEDataSet
SafeCallException	public	TComponent
SetFields	public	TDataSet
Translate	public	TDataSet
UpdateCursorPos	public	TDataSet
UpdateRecord	public	TDataSet
UpdateStatus	public	TBDEDataSet

ApplyUpdates

Writes a dataset's pending cached updates to the database.

```
procedure ApplyUpdates;
```

Description Call *ApplyUpdates* to write a dataset's pending cached updates to a database. This method passes cached data to the database for storage, but the changes are not committed to the database. An application must explicitly call the database component's *Commit* method to commit the changes to the database if the write is successful, or call the database's *Rollback* method to undo the changes if there is an error.

Following a successful write to the database, and following a successful call to the database's *Commit* method, an application should call the *CommitUpdates* method to clear the cached update buffer.

Note The preferred method for updating datasets is with a call to a database component's *ApplyUpdates* method rather than calling each individual dataset's *ApplyUpdates* method. The database component's *ApplyUpdates* method takes care of committing and rolling back transactions and clearing the cache when the operation is successful.

Example The following procedure illustrates how to apply a dataset's cached updates to a database in response to a button click:

```
procedure ApplyButtonClick(Sender: TObject);
begin
  with CustomerQuery do
  begin
    Database1.StartTransaction;
    try
      ApplyUpdates; {try to write the updates to the database};
      Database1.Commit; {on success, commit the changes};
    except
      Database1.Rollback; {on failure, undo the changes};
      raise; {raise the exception to prevent a call to CommitUpdates!}
    end;
    CommitUpdates; {on success, clear the cache}
  end;
end;
```

See also *CachedUpdates* property, *UpdatesPending* property, *CommitUpdates* method, *TDatabase.ApplyUpdates* method, *TDatabase.Commit* method, *TDatabase.Rollback* method

BookmarkValid

Tests the validity of a specified bookmark.

```
type
  TBookmark = Pointer;

function BookmarkValid(Bookmark: TBookmark): Boolean; override;
```

Description Call *BookmarkValid* to determine if a specified bookmark is currently assigned a value. *Bookmark* specifies the bookmark to test.

BookmarkValid returns *True* if a bookmark is valid. Otherwise, it returns *False*.

See also *CompareBookmarks* method, *TTable* object

Cancel

Cancels modifications to the current record if those changes are not yet posted.

```
procedure Cancel;
```

Description Call *Cancel* to undo modifications made to one or more fields belonging to the current record. As long as those changes are not already posted to the database, *Cancel* returns the record to its previous state, and sets the dataset state to *dsBrowse*.

Typically *Cancel* is used to back out of changes in response to user request, or in field validation routines that back out illegal field values. The *TDBNavigator* object contains a Cancel button that triggers a call to *Cancel*.

See also *State* property, *Post* method, *TDBNavigator* object

CancelUpdates

Clears all pending cached updates from the cache and restores the dataset its prior state.

```
procedure CancelUpdates;
```

Description Call *CancelUpdates* to clear all pending cached updates from the cache and restore the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied to the database.

When a dataset is closed, or the *CachedUpdates* property is set to *False*, *CancelUpdates* is called automatically.

Note To undo changes to a single record, call *RevertRecord*.

See also *CachedUpdates* property, *State* property, *ApplyUpdates* method, *RevertRecord* method.

CommitUpdates

Clears the cached updates buffer.

```
procedure CommitUpdates;
```

Description Call *CommitUpdates* to clear the cached updates buffer after both a successful call to *ApplyUpdates* and a database component's *Commit* method. Clearing the cache after applying updates ensures that the cache is empty except for records that could not be processed and were skipped by the *OnUpdateRecord* or *OnUpdateError* event handlers. An application can attempt to modify the records still in the cache.

Record modifications made after a call to *CommitUpdates* repopulate the cached update buffer and require a subsequent call to *ApplyUpdates* to move them to the database.

Note Applications that use a database component's *ApplyUpdates* method to apply and commit pending updates for all datasets associated with the database component do not need to call *CommitUpdates*.

Example The following procedure illustrates how to apply a dataset's cached updates to a database and clear the cached updates buffer in response to a button click:

```
procedure ApplyButtonClick(Sender: TObject);
begin
  with CustomerQuery do
  begin
    Database1.StartTransaction;
    try
      ApplyUpdates; {try to write the updates to the database};
      Database1.Commit; {on success, commit the changes};
    except
      Database1.Rollback; {on failure, undo the changes};
      raise; {raise the exception to prevent a call to CommitUpdates!}
    end;
    CommitUpdates; {on success, clear the cache}
  end;
end;
```

See also *CachedUpdates* property, *UpdatesPending* property, *OnUpdateError* event, *OnUpdateRecord* event, *ApplyUpdates* method, *TDatabase.ApplyUpdates* method

CompareBookmarks

Determines the relationship between two bookmarks.

```
type
  TBookmark = Pointer;

function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
```

Description Call *CompareBookmarks* to determine if two bookmarks are identical or not. *Bookmark1* and *Bookmark2* are the bookmarks to compare.

If the bookmarks differ, *CompareBookmarks* returns *1*. If the *Bookmarks* are identical, or both bookmarks are **nil**, *CompareBookmarks* returns *0*.

See also *BookmarkValid* method, *TTable* object

Create

Creates an instance of a *TBDEDataSet* component.

```
constructor Create(AOwner: TComponent) override;
```

Description Call *Create* to instantiate a dataset component at runtime. Ordinarily applications instantiate *TBDEDataSet* descendants, such as *TTable*, *TQuery*, or

TStoredProc, and these instantiated objects are automatically handled by Delphi.

In some cases, a database application may have a need to create an instance of a *TBDEDataSet* at runtime. *Create*

- Calls its inherited *Create* method.
- Set the *Locale* property of the dataset to the locale of the default session.
- Sets the *CacheBlobs* property to *True*.

See also *CacheBlobs* property, *Destroy* method, *Locale* property, *TQuery* object, *TStoredProc* object, *TTable* object

Destroy

Destroys the instance of a dataset component.

```
destructor Destroy;
```

Description Do not call *Destroy* directly in an application. Usually destruction of datasets is handled automatically by Delphi. If an application creates its own instances of a dataset, however, the application should call *Free*, which verifies that the dataset component is not already freed before calling *Destroy*.

Destroy calls its inherited *Destroy* destructor, if necessary, frees the resources allocated for the BDE asynchronous callback, and sets the *UpdateObject* property for the dataset to **nil**.

See also *Create* method, *UpdateObject* property

DisableConstraints

Disables application of server constraints on the dataset.

```
procedure DisableConstraints;
```

Description Call *DisableConstraints* to prevent, on a temporary basis, application of server constraints replicated to the dataset. For example, when an application iterates through a dataset, it may improve performance to disable constraints temporarily.

Note Calling *DisableConstraints* increments a reference count. As long as this reference count is greater than zero, constraints are disabled for the dataset. To prevent accidental disabling of constraints, always group a call to *DisableConstraints* with a call to *EnableConstraints*.

See also *EnableConstraints* method

EnableConstraints

Enables application of server constraints against dataset entries.

```
procedure EnableConstraints
```

Description Call *EnableConstraints* to turn on application server constraints previously disabled by a call to *DisableConstraints*.

Note Calling *EnableConstraints* decrements a reference count. When this reference count is zero constraints are enabled for the dataset. To prevent accidental disabling of constraints, always follow a call to *DisableConstraints* with a matching call to *EnableConstraints*.

See also *DisableConstraints* method

FetchAll

Retrieves all records from the current cursor position to the end of the file and stores them locally.

```
procedure FetchAll;
```

Description Call *FetchAll* to reduce network traffic when using cached updates. *FetchAll* calls *CheckBrowseMode* to post any pending changes, and then retrieves all records from the current cursor position to the end of the file, and store them locally.

Ordinarily when cached updates are enabled, a transaction retrieves only as much data as it needs for display purposes. Each new fetch starts a new, read-only transaction. To consolidate transactions and reduce network traffic, an application can call *FetchAll* to retrieve all data in a single transaction.

Note Using *FetchAll* is not always appropriate. For example, when an application accesses a database used by many simultaneous clients and there is a high degree of contention for updating the same records, fetching all records at once may not be advantageous because some fetched records may be changed by other applications. Always weigh the advantages of reduced network traffic against the need for reduced record contention.

See also *CachedUpdates* property, *Filter* property, *Filtered* property, *OnFilterRecord* event, *OnUpdateRecord* event

GetCurrentRecord

Retrieves the current record into a buffer.

```
function GetCurrentRecord(Buffer: PChar): Boolean;
```

Description Most applications should not need to call *GetCurrentRecord*. *TDataSet* automatically allocates a buffer for the active record.

Call *GetCurrentRecord* to copy the current record into a buffer allocated by the application. Buffer must be at least as big as the record size indicated by the *RecordSize* property.

See also *RecordSize* property, *ActiveBuffer* method,

GetIndexInfo

Retrieves information about the current index into the index data fields of the dataset.

```
procedure GetIndexInfo;
```

Description Call *GetIndexInfo* to update information about the current index. Ordinarily an application does not need to call *GetIndexInfo* because this routine is used internally to retrieve index information as needed. Some applications, however, may want to ensure that the index information used by the dataset is up-to-date.

GetIndexInfo queries the BDE for index information, including

- Whether or not the current index is a dBASE expression index.
- Whether or not the index is case insensitive.
- The number of fields that make up the key for the index.
- The field map for the index.
- The size of the index key.

Number of fields and field map are not retrieved if the index is a dBASE expression index.

See also *ExpIndex* property

Locate

Searches the dataset for a specified record and makes that record the current record.

```
type
  TLocateOption = (loCaseInsensitive, loPartialKey);
  TLocateOptions = set of TLocateOption;

function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean;
```

Description Call *Locate* to search a dataset for a specific record and position the cursor on it. *KeyFields* is a string containing a semicolon-delimited list of field names on which to search.

KeyValues is a variant array containing the values to match in the key fields. To specify multiple search values, pass *KeyValues* as a variant array as an argument, or construct a variant array on the fly using the *VarArrayOf* routine. For example,

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver', 'P']),
[loPartialKey]);
```

Options is a set that optionally specifies additional search latitude when searching on string fields. If *Options* contains the *loCaseInsensitive* setting, then *Locate* ignores case when matching fields. If *Options* contains the *loPartialKey* setting, then *Locate* finds the first record that fulfills at least some

initial part of the *KeyValues* criteria for record matching. If *Options* is an empty set, or if the *KeyFields* are not string fields, *Options* is ignored.

Locate returns *True* if it finds a matching record, and makes that record the current one. Otherwise *Locate* returns *False*.

Locate uses the fastest possible method to locate matching records. If the search fields in *KeyFields* are indexed and the index is compatible with the specified search options, *Locate* uses the index. Otherwise *Locate* creates a filter for the search.

See also *FindField* method, *Lookup* method

Lookup

Retrieves field values from a record that matches specified search values.

```
function Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant;
```

Description Call *Lookup* to retrieve values for specified fields from a record that matches search criteria. *KeyFields* is a string containing a semicolon-delimited list of field names on which to search.

KeyValues is a variant array containing the values to match in the key fields. To specify multiple search values, pass *KeyValues* as a variant array as an argument, or construct a variant array on the fly using the *VarArrayOf* routine.

ResultFields is a string containing a semicolon-delimited list of field names whose values should be returned from the matching record.

Lookup returns a variant array containing the values from the fields specified in *ResultFields*.

Lookup uses the fastest possible method to locate matching records. If the search fields in *KeyFields* are indexed, *Lookup* uses the index. Otherwise *Lookup* creates a filter for the search.

See also *Locate* method

RevertRecord

Restores the current record in the dataset to an unmodified state when cached updates are enabled.

```
procedure RevertRecord;
```

Description Call *RevertRecord* to undo changes made to the current record when cached updates are enabled. With minor coding, *RevertRecord* can be used to undelete records when cached updates are enabled, as the following procedure demonstrates:

```
procedure UndeleteAll(DataSet: TDataSet);
begin
```

```
  with DataSet do
  begin
    UpdateRecordTypes := [rtDeleted]; {make only deleted records visible}
    try
      First; {move to beginning of dataset}
      while not EOF do
      begin
        RevertRecord; {undelete the current record}
       Next; {move to the next record}
      end;
    finally
      UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
    end;
  end;
end;
```

Note To undo all changes to all pending updates in the cache, call *CancelUpdates*.

See also *CancelUpdates* method

UpdateStatus

Reports the current update status for the dataset.

```
type
  TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);

function UpdateStatus: TUpdateStatus;
```

Description Call *UpdateStatus* to determine the current update status for the dataset when cached updates are enabled. Update status can change frequently as records are edited, inserted, or deleted. *UpdateStatus* offers a convenient method for applications to assess the current status before undertaking or completing operations that depend on the update status of the dataset.

See also *CachedUpdates* property, *ApplyUpdates* method, *CancelUpdates* method, *CommitUpdates* method, *RevertRecord* method

TBevel

Unit: ExtCtrls | Palette: Additional

TObject • TPersistent • TComponent • TControl • TGraphicControl • TBevel

TBevel represents a beveled outline.

Description Use *TBevel* to create beveled boxes, frames, or lines. The bevel can appear raised or lowered.

See also *TShape* object

Properties

The following table lists the properties of *TBevel*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
BoundsRect	public		TControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TControl
ClientRect	public, read-only		TControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Height	published		TControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentShowHint	published	True	TControl
Shape	published	bsBox	TBevel
ShowHint	published		TControl
Style	published	bsLowered	TBevel
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

Shape

Shape determines the shape of the bevel.

```
type TBevelShape = (bsBox, bsFrame, bsTopLine, bsBottomLine, bsLeftLine, bsRightLine);
```

```
property Shape: TBevelShape;
```

Description Set Shape to specify whether the bevel appears as a line, a box, or a frame. These are the possible values of *Shape*:

Value	Meaning
bsBox	The entire client area appears raised or lowered, depending on the value of *Style*.
bsFrame	The client area is outlined by a raised or lowered frame.
bsTopLine	The bevel displays a line at the top of the client area.
bsBottomLine	The bevel displays a line at the bottom of the client area.
bsLeftLine	The bevel displays a line at the left side of the client area.
bsRightLine	The bevel displays a line at the right side of the client area.

See also *ClientRect* property, *Style* property

Style

Style determines whether the bevel appears raised or lowered.

```
type TBevelStyle = (bsLowered, bsRaised);
```

```
property Style: TBevelStyle;
```

Description Set *Style* to indicate whether the bevel should create a raised or a lowered effect. When the *Shape* property is *bsBox*, the entire client area appears raised or lowered. For all other values of *Shape*, the bevel displays a raised or lowered line along the edge or edges of the client area. These are the possible values of *Style*:

Value	Meaning
bsLowered	The bevel is lowered.
bsRaised	The bevel is raised.

See also *Shape* property

Methods

The following table lists the methods of *TBevel*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject

Method	Visibility	Documented in
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
Create	public	TBevel
DefaultHandler	public	TObject
Destroy	public	TGraphicControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTextBuf	public	TControl
GetTextLen	public	TControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
Invalidate	public	TControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Perform	public	TControl
Refresh	public	TControl
RemoveComponent	public	TComponent
Repaint	public	TControl
SafeCallException	public	TComponent
ScreenToClient	public	TControl

Method	Visibility	Documented in
SendToBack	public	TControl
SetBounds	public	TControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TControl

Create

Create creates and initializes an instance of *TBevel*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a bevel at runtime. Bevels placed on forms at design time are created automatically. After calling the inherited constructor, *Create* initializes *Style* to *bsLowered* and *Shape* to *bsBox*.

See also *Shape* property, *Style* property

TBinaryField

Unit DB

TObject • TPersistent • TComponent • TField • TBinaryField

TBinaryField represents an untyped binary field in a dataset.

Description *TBinaryField* is an abstract object that encapsulates the fundamental behavior common to untyped binary fields. In addition to the properties, methods, and events common to all fields that are introduced by the *TField* class, *TBinaryField* reimplements the properties that are needed to read and write binary data from or to the underlying database table.

Do not create instances of *TBinaryField*. *TBinaryField* descendants are created automatically each time a dataset is activated.

See also *TBytesField* object, *TVarBytesField* object

Properties

The following table lists the properties of *TBinaryField:*

Property	Visibility	Default	Documented in
Alignment	published	taLeftJustify	TField
AsBoolean	public		TField
AsCurrency	public		TField
AsDateTime	public		TField

Property	Visibility	Default	Documented in
AsFloat	public		TField
AsInteger	public		TField
AsString	public		TBinaryField
AsVariant	public		TBinaryField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
CurValue	public, read-only		TField
CustomConstraint	published		TField
DataSet	public		TField
DataSize	public, read-only		TField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditMask	public		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
HasConstraints	published, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField

Property	Visibility	Default	Documented in
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	published	16	TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TBinaryField
VCLComObject	public		TComponent
Visible	published	True	TField

AsString

AsString represents the value of a binary field as a string.

property AsString: string;

Description Use *AsString* to read or write the value of the field as a string. Delphi strings are not limited to text, so the value represented by *AsString* is the same as the unformatted binary value of the field. When setting the value of *AsString*, if the assigned value is longer than the *DataSize* of the field, it will be truncated to include only *DataSize* bytes.

Note: Reading the *Text* property of a binary field gives the field name of the field, not the value of *AsString*.

See also *DataSize* property, *Value* property

AsVariant

AsVariant converts the value of a binary field to or from a variant array of *Byte* values.

property AsVariant: Variant;

Description Use *AsVariant* to read or write the value of the field as a variant array of byte values. For example, this code assigns a variant array of bytes to a binary field value:

```
var
  V: Variant;
begin
  V := VarArrayCreate([0, BinaryField.DataSize - 1], VarByte);
```

```
    V[i] := 1;
    :
    BinaryField.AsVariant := V;
  end;
```

To assign a binary field value to a variant variable, use code similar to this:

```
VariantVariable:= MyBinaryField.AsVariant;
```

Because binary fields are stored in their database tables as unformatted arrays of bytes, the *AsVariant* property for a binary field is the same as the *Value* property.

See also *Value* property

Value

Value is the actual data in the binary field.

property Value: variant;

Description Use *Value* to read data directly from and write data directly to a binary field component at runtime. For binary fields, the *Value* property is equivalent to the *AsVariant* property.

See also *AsVariant* property

Events

The following table lists the events of *TBinaryField:*

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TBinaryField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject

Method	Visibility	Documented in
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TBinaryField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create creates an instance of *TBinaryField*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TBinaryField*. *Create* simply calls the inherited *Create* method of *TField*.

Most applications do not need to call *Create* directly. Binary fields are automatically created by the dataset that contains them.

See also *TField.Create* method

TBitBtn

Unit: Buttons | Palette: Additional

TObject • TPersistent • TComponent • TControl • TWinControl • TButtonControl • TButton • TBitBtn

TBitBtn is a push button control that can include a bitmap on its button face.

Description Bitmap buttons have the general behavior of button controls. Use them to initiate actions from forms and dialog boxes.

Bitmap buttons have additional properties used to specify the bitmap images, their appearance and placement on the button. You can choose from predefined bitmap buttons styles or use your own, customized bitmap for the button. Although the button can be associated with only a single bitmap, the bitmap (*glyph* property) can be subdivided into four equally sized portions which display based on the state of the button: up, down, and disabled (note that speed buttons can make use of the fourth part of the glyph with their fourth state called "stay down").

The *Kind* property of *TBitBtn* lets you easily create buttons that have common behaviors, such as Ok, Cancel, Help, and so on. These predefined button types have appropriate graphical images and default behaviors, so you can easily add common button types to your application without being distracted with the coding overhead.

See also *TButton* object, *TSpeedButton* object, *kind* property

Properties

The following table lists the properties of *TBitBtn*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Cancel	published	False	TButton
Caption	published		TControl
ClientHeight	public		TControl

Property	Visibility	Default	Documented in
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
Default	published		TButton
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Font	published		TControl
Glyph	published		TBitBtn
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Kind	published	bkCustom	TBitBtn
Layout	published	blGlyphLeft	TBitBtn
Left	published		TControl
Margin	published	–1	TBitBtn
ModalResult	published	0	TButton
Name	published		TControl
NumGlyphs	published	1	TBitBtn
Owner	public, read-only		TComponent
Parent	public		TControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ShowHint	published		TControl
Showing	public, read-only		TWinControl
Spacing	published	4	TBitBtn
Style	published	bsAutoDetect	TBitBtn

Property	Visibility	Default	Documented in
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

Glyph

Specifies the bitmap that appears on the selected bitmap button.

property Glyph: TBitmap;

Description Use the Open dialog box that appears as an editor in the Object Inspector to choose a bitmap file (with a .BMP extension) to use on the button, or specify a bitmap file at runtime.

You can provide up to four images on a bitmap button or speed button with a single bitmap. Delphi then displays one of these images depending on the state of the button.

Image position in bitmap	Button state	Description
First	Up	This image appears when the button is up; the normal state of the button This image is also used when the button has focus (for example, if you tab to it). In this case, a focus rectangle is drawn around the button. If no other images exist in the bitmap, Delphi also uses this image for all other images.
Second	Disabled	This image usually appears dimmed to indicate that the button can't be selected.
Third	Down	This image appears when a button is clicked. The up state image reappears when the user releases the mouse button.

If only one image is present, Delphi attempts to represent the other states by altering the image slightly for the different states, although the stay down state is always the same as the up state. If you aren't satisfied with the results, you can provide one or more additional images in the bitmap.

If you have multiple images in a bitmap, you must specify the number of images that are in the bitmap with the *NumGlyphs* property. All images must be the same size and next to each other in a horizontal row.

Note The lower left hand pixel of the bitmap is reserved for the "transparent" color. Any pixel in the bitmap which matches that lower left hand pixel will be "transparent."

See also *Kind* property, *NumGlyphs* property

Kind

Determines the kind of bitmap button.

```
type TBitBtnKind = (bkCustom, bkOK, bkCancel, bkHelp, bkYes, bkNo, bkClose, bkAbort,
  bkRetry, bkIgnore, bkAll);

property Kind: TBitBtnKind;
```

Description The *TBitBtnKind* type contains the values the *Kind* property of a *TBitBtn* bitmap button can assume.

These are the possible values and their meanings:

Value	Meaning
bkCustom	You indicate which bitmap you want the bitmap button to have by setting the value of the *Glyph* property to the bitmap of your choice. Like push buttons, you can either select a *ModalResult* for the button, or you can supply the code to respond to an *OnClick* event.
bkOK	A green check mark and the text "OK" appear on the button face. The button becomes the default button (the *Default* property is automatically set to *True*). When the user chooses the button, the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrOK*.
bkCancel	A red X and the text "Cancel" appear on the button face. The button becomes the Cancel button (the *Cancel* property is automatically set to *True*). When the user chooses the button, the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrCancel*.
bkYes	A green check mark and the text "Yes" appear on the button face. The button becomes the default button (the *Default* property is automatically set to *True*). When the user chooses the button, any changes the user made in the dialog box are accepted and the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrYes*.
bkNo	A red no symbol and the text "No" appear on the button face. The button becomes the Cancel button (the *Cancel* property is automatically set to *True*). When the user chooses the button, any changes the user made in the dialog box are canceled and the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrNo*.
bkHelp	A cyan question mark and the text "Help" appear on the button face. When the user chooses the button, a Help screen in the application's Help file appears. The Help file that appears is the file specified as the value of the application's *HelpFile* property. The value of the *HelpContext* property of the button specifies which Help screen in the Help file appears.
bkClose	A door with a green exit sign over it (use your imagination) and the text "Close" appear on the button face. When the user chooses the button, the form closes. The *Default* property of the button is *True*.
bkAbort	A red X and the text "Abort" appear on the button face.
bkRetry	A green circular arrow and the text "Retry" appear on the button face.

Value	Meaning
bkIgnore	A green man walking away and the text "Ignore" appear on the button face. Use it to allow the user to continue after an error has occurred.
bkAll	A double green check mark and the text "Yes to All" appear on the button face.

See also *Glyph* property, *Style* property

Layout

Determines where the image appears on the bitmap button.

```
type TButtonLayout = (blGlyphLeft, blGlyphRight, blGlyphTop, blGlyphBottom);
property Layout: TButtonLayout;
```

Description The *TButtonLayout* type defines the values the *Layout* property of a bitmap button (*TBitBtn*). These are the possible values:

Value	Meaning
blGlyphLeft	The image appears at the left side of the text or caption.
blGlyphRight	The image appears at the right side of the text or caption.
blGlyphTop	The image appears at the top of the text or caption.
blGlyphBottom	The image appears at the bottom of the text or caption.

See also *Glyph* property, *Kind* property, *Margin* property

Margin

Determines the number of pixels between the edge of the image (specified in the *Glyph* property) and the edge of the button.

```
property Margin: Integer;
```

Description The edges that the margin separates depends on the layout of the image and text (specified in the *Layout* property). For example, if *Layout* is *blGlyphLeft*, the margin appears between the left edge of the image and the left edge of the button. If *Margin* is 3, three pixels separates the image and the button edges. If *Margin* is 0, no distance in pixels separates the image and the button edges.

If *Margin* is –1 (which it is by default), then the image and text (specified in the *Caption* property) are centered. The number of pixels between the image and button edge is equal to the number of pixels between the opposite edge of the button and the text.

See also *Glyph* property, *Layout* property, *Spacing* property

NumGlyphs

Indicates the number of images that are in the graphic specified in the *Glyph* property for use on a bitmap button or speed button.

```
type TNumGlyphs: 1..4;
```

```
property NumGlyphs: TNumGlyphs;
```

Description If you have multiple images in a bitmap, you must specify the number of images that are in the bitmap with the *NumGlyphs* property. All images must be the same size and next to each other in a row. Valid *NumGlyphs* values are 1 to 4. The default value is 1.

You can provide up to four images on a bitmap button or speed button with a single bitmap. Delphi then displays one of these images depending on the state of the button. Only one image is required in a bitmap.

Image position in bitmap	Speed button state	Description
First	Up	This image appears when the button is unselected. If no other images exist in the bitmap, Delphi uses this image for all other images.
Second	Disabled	This image usually appears dimmed and indicates that the button can't be selected.
Third	Down	This image appears when a button is clicked. The up state image then reappears when the user releases the mouse button.

If only one image is present, Delphi attempts to represent the other states by altering the image slightly for the different states, although the stay down state is always the same as the up state. If you aren't satisfied with the results, you can provide additional images in the bitmap.

The *TNumGlyphs* type defines the range of values (1-4) the *NumGlyphs* property of a bitmap button (*TBitBtn*) can assume.

See also *Glyph* property

Spacing

Determines where the image and text appear on a bitmap or speed button.

```
property Spacing: Integer;
```

Description Spacing determines the number of pixels between the image (specified in the *Glyph* property) and the text (specified in the *Caption* property). The default value is 4 pixels.

If *Spacing* is a positive number, its value is the number of pixels between the image and text. If *Spacing* is 0, no pixels will be between the image and text. If *Spacing* is –1, the text appears centered between the image and the button

edge. The number of pixels between the image and text is equal to the number of pixels between the text and the button edge opposite the glyph.

See also *Glyph* property, *Layout* property, *Margin* property

Style

Determines the appearance of a bitmap button.

```
type TButtonStyle = (bsAutoDetect, bsWin31, bsNew);
property Style: TButtonStyle;
```

Description The *TButtonStyle* type contains the values the *Style* property of bitmap buttons can assume.

These are the possible values:

Value	Meaning
bsAutoDetect	When you are using Windows 3.x, the bitmap button uses the standard Windows 3.x look. When you are using a later version of Windows, the bitmap button uses a newer look.
bsWin31	Uses the standard Windows 3.1 look, regardless of which version of Windows you are running.
bsNew	Uses a new bitmap button look, regardless of which version of Windows you are running.

See also *Kind* property

Events

The following table lists the events of *TBitBtn*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TBitBtn*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Click	public	TBitBtn
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TBitBtn
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TBitBtn
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent

Method	Visibility	Documented in
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Click

Simulates a mouse click, as if the user had clicked the button.

procedure Click;

Description When *Click* is called any code attached to the *OnClick* event executes.

If the value of the *Kind* property is *bkClose*, calling *Click* closes the form. If the value of *Kind* is *bkHelp*, calling *Click* displays the Help screen assigned with the *HelpContext* property to the bitmap button. If *Kind* has any other value, the inherited *Click* method is called.

See also *OnClick* event

Create

Create constructs an instance of the button control and initializes its values.

constructor Create(AOwner: TComponent); **override**;

Description Use *Create* to programmatically instantiate a button. For buttons created in the Delphi form designer, *Create* is called automatically.

Create

- Calls the inherited *Create* method.
- Creates the button glyph.
- Assigns the *GlyphChanged* method to the *OnChange* event.
- Creates a *TCanvas* object for the bitmap button.
- Sets the *Style* property value to *bsAutoDetect*.
- Sets the *Kind* property value to *bkCustom*.
- Sets the *Layout* property value to *blGlyphLeft*.
- Sets the *Spacing* property value to 4.
- Sets the *Margin* property value to –1.

See also *TWinControl.Create* method, *Destroy* method

;Destroy

Destroy is the destructor for the bitmap button object.

destructor Destroy; **override**;

Description *Destroy* frees the *TGlyph* and *TCanvas* objects before calling the inherited *Destroy* method.

See also *Create* method

TBlobField

Unit: DB

TObject•TPersistent•TComponent•TField•TBlobField

TBlobField represents a field in a dataset that holds a reference to a binary large object (BLOB).

Description BLOB fields are database fields that contain binary data of arbitrary length. Unlike binary fields, BLOB fields do not store the binary data directly in the

database table. Instead, the field in the physical database table contains a reference to a separate file that contains the individual BLOB value for the field.

TBlobField introduces new methods to support streaming of the data in the BLOB field.

As a descendant of *TField*, *TBlobField* includes many properties, methods, and events that are useful for managing the value and properties of a field in a database.

BLOB field components can represent different arbitrarily large data types. These data types are distinguished in the header of the binary data. In addition to the field types supported directly, *TBlobField* is the direct ancestor of two BLOB field components: *TMemoField* (*ftMemo*) and *TGraphicField* (*ftGraphic*).

See also *TBinaryField* object, *TBlobStream* object, *TGraphicField* object, *TMemoField* object

Properties

The following table lists the properties of *TBlobField*:

Property	Visibility	Default	Documented in
Alignment	published	taLeftJustify	TField
AsBoolean	public		TField
AsCurrency	public		TField
AsDateTime	public		TField
AsFloat	public		TField
AsInteger	public		TField
AsString	public		TBlobField
AsVariant	public		TBlobField
AttributeSet	public		TField
BlobSize	public, read-only		TBlobField
BlobType	published		TBlobField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
CurValue	public, read-only		TField
CustomConstraint	published		TField

Property	Visibility	Default	Documented in
DataSet	public		TField
DataSize	public, read-only		TField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditMask	public		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
HasConstraints	published, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TBlobField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField
Modified	public		TBlobField
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	published	0	TField
Tag	published	0	TComponent
Text	public		TField
Transliterate	public		TBlobField
ValidChars	public		TField
Value	public		TBlobField

Property	Visibility	Default	Documented in
VCLComObject	public		TComponent
Visible	published	True	TField

AsString

AsString converts the value of the BLOB field to or from a string value.

property AsString: **string**;

Description Use *AsString* to assign the BLOB field value to a string or to assign a string value to the BLOB field.

For example, this code assigns a string value to a BLOB field value:

```
MyBLOBField.AsString := 'Assign the string to a BLOB field';
```

To assign a BLOB field value to a string value, use code similar to this:

```
BlobString := MyBLOBField.AsString;
```

Typically, using *AsString* makes sense only if the BLOB contains text, such as that in a memo field component. The string data type can, however, store binary data as well. Thus, even BLOB fields for nontextual BLOB types such as *ftGraphic* or *ftTypedBinary* can use the *AsString* property.

Use the *AsString* property rather than the *Text* property to set the BLOB field's value to textual data. Trying to set the *Text* property will raise an exception.

See also *AsVariant* property, *BlobType* property, *SetText* method, *Text* property, *Value* property

AsVariant

AsVariant converts the value of the BLOB field to or from a variant value.

property AsVariant: Variant;

Description Use *AsVariant* to

- Assign the BLOB field value to a variant variable.
- Assign a variant value to the BLOB field.

Using variants is useful when you don't know the actual field type and don't want to use one of the other *As...* properties to convert the field value to another type.

See also *AsString* property, *Value* property

BlobSize

BlobSize is the number of bytes of data contained in the blob field on the current record.

```
property BlobSize: Integer;
```

Description Read *BlobSize* to determine the amount of memory required to store the contents of the BLOB field. Use *BlobSize* instead of the *DataSize* property, which always returns 0 for BLOB fields.

Note Unlike the *DataSize* property, which specifies the size for field data on any record, *BlobSize* gives the size of the BLOB data for the current record only.

See also *DataSize* property

BlobType

BlobType identifies the type of the BLOB field associated with a BLOB field object.

```
type TBlobType = ftBlob..ftTypedBinary;
```

```
property BlobType: TBlobType;
```

Description Use *BlobType* to set or read the data type for the BLOB data associated with the field component.

Specify *BlobType* when using a *TBlobField* object to represent a BLOB field type that isn't directly supported (such as a formatted memo BLOB). The *BlobType* ensures that the BLOB type in the table matches the expected field type.

TBlobType is a range of *TFieldType* values. These are the values in that range:

Value	Description
ftBlob	Binary Large Object field
ftMemo	Text memo field
ftGraphic	Bitmap field
ftFmtMemo	Formatted text memo field
ftParadoxOle	Paradox OLE field
ftDBaseOle	dBase OLE field
ftTypedBinary	Typed binary field

See also *DataType* property, *SetFieldType* method

IsNull

IsNull indicates whether the BLOB field contains any data.

```
property IsNull: Boolean;
```

Description Read *IsNull* to determine whether the BLOB field is blank. *IsNull* is *True* if the BLOB field is empty. *IsNull* is *False* if the BLOB field contains any data.

See also *Clear* method, *Value* property

Modified

Modified indicates whether a the value for the BLOB field was changed since it was last read from disk.

property Modified: Boolean;

Description Set *Modified* to *True* when changing the value to a BLOB field using the *SetData* method or direct Borland Database Engine (BDE) API calls. If the value of the BLOB field is set by using the properties of *TBlobField*, or by using a *TBlobStream* object, *Modified* is automatically set to *True*.

BLOB fields use the *Modified* property to trigger data events and to indicate when the BDE needs to clean up the buffers used to store BLOB data.

See also *AsString* property, *AsVariant* property, *SetData* method, *TBlobStream* object, *Value* property

Transliterate

Transliterate determines whether any text in the BLOB field should be converted between the ANSI character set and the character set identified by the dataset.

property Transliterate: Boolean;

Description Use *Transliterate* when the physical database table identified by the dataset does not use an ANSI language driver and the data may contain extended ASCII characters.

When *Transliterate* is *True*, BLOB fields call the *AnsiToNative* function to translate the ANSI characters to the character set used by the dataset when the value is changed using the *Value* property, the *AsString* property, or the *AsVariant* property. When these properties are read to obtain the value of the BLOB field, the *NativeToAnsi* function translates the characters in the database table to the ANSI character set.

Note *TBlobStream* objects also check the *Transliterate* property when reading from or writing to a BLOB field.

See also *AnsiToNative* function, *AsString* property, *AsVariant* property, *DataSet* property, *NativeToAnsi* procedure, *TBlobStream* object, *TDataSet.Locale* property, *Value* property

Value

Value is the actual data in the BLOB field.

property Value: **string**;

Description Use *Value* to read data directly from and write data directly to a BLOB field component at runtime. For BLOB fields, the *Value* property is equivalent to the *AsString* property.

Note The string data type can store binary data as well as text. Thus, even BLOB fields for nontextual BLOB types such as *ftGraphic* or *ftTypedBinary* can get or set their data using a string.

See also *AsString* property, *BlobType* property

Events

The following table lists the events of *TBlobField*:

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TBlobField:*

Method	Visibility	Documented in
Assign	public	TBlobField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TBlobField
Create	public	TBlobField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField

Method	Visibility	Documented in
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TBlobField
IsValidChar	public	TField
LoadFromFile	public	TBlobField
LoadFromStream	public	TBlobField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SaveToFile	public	TBlobField
SaveToStream	public	TBlobField
SetData	public	TField
SetFieldType	public	TBlobField
SetText	public	TBlobField
Validate	public	TField

Assign

Assign copies a value to the BLOB field.

procedure Assign(Source: TPersistent);

Description Use *Assign* to copy data to the BLOB field. *Assign* copies the *Value* of a BLOB field from

- Another BLOB field
- A *TStrings* object
- A *TBitmap* object, if the *BlobType* is *ftGraphic* or *ftTypedBinary*
- A *TPicture* object, if the *BlobType* is *ftGraphic* or *ftTypedBinary*
- Any object that supports *TBlobField* in its *AssignTo* method

See also *TBitmap* object, *TPicture* object, *TStrings* object, *TPersistent.AssignTo* method, *Value* property

Clear

Clear clears the current value in the BLOB field.

```
procedure Clear;
```

Description Use *Clear* to delete the current value in the field and leave the field without a value.

See also *IsNull* property, *Required* property, *Value* property

Create

Create creates an instance of a BLOB field component.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to create and initialize an instance of *TBlobField*. *Create* sets the *DataType* to *ftBlob* after calling the inherited constructor.

Explicit calls to *Create* are seldom needed because the BLOB field component is instantiated automatically for all BLOB fields in a dataset.

See also *BlobType* property, *DataType* property, *TField.Create* method

IsBlob

IsBlob overrides the inherited method to indicate that this is a BLOB field.

```
class function IsBlob: Boolean; override;
```

Description Use *IsBlob* when working with a *TField* object to determine whether it represents BLOB data. For example, before using the *GetData* or *SetData* method, call *IsBlob* to determine whether the method can be used. The *IsBlob* method for *TBlobField* always returns *True*.

See also *GetData* method, *SetData* method

LoadFromFile

LoadFromFile loads a BLOB from a file into the field.

```
procedure LoadFromFile(const FileName: string);
```

Description Use *LoadFromFile* to load the contents of a file into a BLOB field. Specify the name of the file to load into the field as the value of the *FileName* parameter.

See also *LoadFromStream* method, *SaveToFile* method

LoadFromStream

LoadFromStream loads a BLOB from a stream into the field.

```
procedure LoadFromStream(Stream: TStream);
```

Description Use *LoadFromStream* to copy the contents of a stream into the BLOB field. Specify the stream from which the field's value is copied as the value of the *Stream* parameter.

See also *LoadFromFile* method, *SaveToStream* method

SaveToFile

SaveToFile saves the contents of the BLOB field to a file.

```
procedure SaveToFile(const FileName: string);
```

Description Use *SaveToFile* to save the contents of the BLOB field to a file. Specify the name of the file as the value of the *FileName* parameter.

See also *LoadFromFile* method, *SaveToStream* method

SaveToStream

SaveToStream saves the contents of the BLOB field to a stream.

```
procedure SaveToStream(Stream: TStream);
```

Description Use *SaveToStream* to copy the contents of a BLOB field to a stream. Specify the name of the stream to which the field's value is saved as the value of the *Stream* parameter.

See also *LoadFromStream* method, *SaveToFile* method

SetFieldType

SetFieldType specifies the type of the BLOB field.

```
procedure SetFieldType(Value: TFieldType); override;
```

Description Use *SetFieldType* to change the field type of a BLOB field, overriding the default type.

Normally, the type of the physical database field is checked and a *TFieldType* is selected that matches most closely. BLOB fields can indicate more explicitly what kind of BLOB is in the database. Specify the BLOB type as the value of the *Value* parameter.

These following table contains the possible values for BLOB fields. Calling *SetFieldType* with any other data type does nothing.

Value	Description
ftBlob	Binary Large Object field
ftMemo	Text memo field
ftGraphic	Bitmap field
ftFmtMemo	Formatted text memo field
ftParadoxOle	Paradox OLE field

Value	Description
ftDBaseOle	dBase OLE field
ftTypedBinary	Typed binary field

Note Calling *SetFieldType* does the same thing as setting the *BlobType* property.

See also *BlobType* property, *DataType* property

SetText

SetText raises an exception for BLOB field objects.

```
procedure SetText(const Value: string);
```

Description *SetText* prevents data-aware controls from storing the '(fieldname)' string back into the dataset. To set the value of a BLOB field to textual data, use the *AsString* property.

See also *AsString* property, *Text* property, *Value* property

TBooleanField

Unit: DB

TObject • TPersistent • TComponent • TField • TBooleanField

TBooleanField represents a field containing Boolean values.

Description Boolean fields can hold values of *True* or *False*.

TBooleanField introduces new properties to convert between boolean values and other data types, and to provide display strings that represent various interpretations of the data. As a descendant of *TField*, *TBooleanField* includes many properties, methods, and events that are useful for managing the value and properties of a field in a database.

See also *DataSet object*

Properties

The following table lists the properties of *TBooleanField*:

Property	Visibility	Default	Documented in
Alignment	published	taLeftJustify	TField
AsBoolean	public		TBooleanField
AsCurrency	public		TField
AsDateTime	public		TField
AsFloat	public		TField

Property	Visibility	Default	Documented in
AsInteger	public		TField
AsString	public		TBooleanField
AsVariant	public		TBooleanField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
CurValue	public, read-only		TField
CustomConstraint	published		TField
DataSet	public		TField
DataSize	public, read-only		TBooleanField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayValues	published		TBooleanField
DisplayWidth	published		TField
EditMask	public		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField
Name	published		TComponent

Property	Visibility	Default	Documented in
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	public		TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TBooleanField
VCLComObject	public		TComponent
Visible	published	True	TField

AsBoolean

AsBoolean reads or writes the value of a Boolean field to or from a Boolean value.

```
property AsBoolean: Boolean;
```

Description Use *AsBoolean* to set or get the value of a Boolean field using the Boolean type. Physical Boolean database fields store boolean values as WordBool values. *AsBoolean* converts between the WordBool value and the Delphi Boolean type.

See also *TField.DataType* property, *Value* property

AsString

AsString converts the value of a Boolean field to or from a string value.

```
property AsString: string;
```

Description Use *AsString* to

- Assign the Boolean field value to string variable.
- Assign a string value to the Boolean field.

For example, this code assigns a string value to an Boolean field value:

```
MyBoolField.AsString := 'TRUE';
```

To assign a Boolean field value to a string variable, use code similar to this:

```
StringVar := MyBoolField.AsString;
```

If the *DisplayValues* property has not been set, when a Boolean value is inserted or updated using the *AsString* property, the value is set to *True* if the

string is 'True', and *False* if the string is 'False'. Any other string raises an *EDatabaseError* exception.

The *DisplayValues* property specifies the strings associated with the values of *True* or *False*. Change the *DisplayValues* property to change the strings that must be assigned to set the value of the Boolean field to *True* or *False*.

Reading the *AsString* property returns the string that corresponds to the value of the field. If *DisplayValues* has not been set, this is 'True' or 'False'.

See also *DisplayValues* property

AsVariant

AsVariant converts the value of a Boolean field to or from a variant value.

property AsVariant: Variant;

Description Use *AsVariant* to

- Assign the Boolean field value to variant variable.
- Assign a variant value to the Boolean field.

For example, this code assigns a variant value to a Boolean field:

```
MyBoolField.AsVariant := VariantVariable;
```

To assign a Boolean field value to a variant value, use code similar to this:

```
VariantVariant:= MyBoolField.AsVariant;
```

Using variants is useful when both the actual field type and the type of the data is unknown.

See also *AsBoolean* property, *AsString* property, *Value* property

DataSize

DataSize is the amount of memory that is needed to store the value of a Boolean field in its native database format.

property DataSize: Word;

Description Check *DataSize* to determine the number of bytes required to store a Boolean field component's value. Use *DataSize* to determine the size needed for a buffer when working with the *GetData* and *SetData* methods. *GetData* and *SetData* work with the field's value in native database format.

For Boolean fields, *DataSize* is always 2, because database tables use the WordBool type to store boolean values. This type is not the same as the Boolean type used by *TBooleanField* to store and manipulate boolean values.

See also *AsBoolean* property, *GetData* method, *SetData* method

DisplayValues

DisplayValues controls how the Boolean field is translated to and from display format.

property DisplayValues: **string**;

Description Use *DisplayValues* to specify strings you want used to represent Boolean values. Use any pair of phrases, separated by a semicolon. For example, to have the *True* and *False* values correspond to the letters T and F, respectively, set *DisplayValues* to T;F. Similarly, to set the values of *True* and *False* to the strings Yes and No, set *DisplayValues* to Yes;No.

The string associated with *True* or *False* can be an empty string. To set the value for *True* to an empty string, set *DisplayValues* to a string that begins with a semicolon. For example, to associate *False* with the string Fail, and *True* with an empty string, set *DisplayValues* to ;Fail. To associate False with an empty string, set *DisplayValues* to the string for *True*, with no semicolon at all.

The strings associated with *True* and *False* by *DisplayValues* appear in data-aware controls when they display the data for a Boolean field. If one of the strings is an empty string, Boolean values associated with that string appear blank in data-aware controls.

These strings are also used when getting or setting the Boolean field's *AsString* property.

See also *AsString* property

Value

Value is the actual data in the Boolean field.

property Value: Boolean;

Description Use *Value* to read data directly from and write data directly to a Boolean field object at runtime. Using *Value* is the same as using the *AsBoolean* property. Use *Value* when you know the field component is a Boolean field. Use *AsBoolean* when working with a generic field component.

See also *AsBoolean* property

Events

The following table lists the events of *TBooleanField:*

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField

Event	Visibility	Documented in
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods on *TBooleanField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TBooleanField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject

Method	Visibility	Documented in
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create instantiates a Boolean field object.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TBooleanField*. After calling the inherited constructor, *Create* sets the *DataType* to *ftBoolean*, and initializes the strings associated with Boolean values to 'True' and 'False'.

It is seldom necessary to explicitly call *Create* because a Boolean field object is instantiated automatically for all Boolean fields in a dataset.

See also *DataType* property, *TField.Create* method

TButton

Unit: StdCtrls | Palette: Standard

TObject • TPersistent • TComponent • TControl • TWinControl • TButtonControl • TButton

TButton is a push button control.

Description Users choose button controls to initiate actions. Buttons are most commonly used in dialog boxes.

See also *TBitBtn* component, *TSpeedButton* component

Properties

The following table lists the properties of *TButton*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Cancel	published	False	TButton
Caption	published		TControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
Default	published		TButton
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
ModalResult	published	0	TButton
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	published	–1	TWinControl

Property	Visibility	Default	Documented in
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

Cancel

Determines whether the button's *OnClick* event handler executes when the *Escape* key is pressed.

property Cancel: Boolean;

Description If *Cancel* is *True*, the button's *OnClick* event handler executes when the user presses *Esc*. Although an application can have more than one Cancel button, the form calls the *OnClick* event handler only for the first visible button in the tab order.

See also *Default* property

Default

Determines whether the button's *OnClick* event handler executes when the *Enter* key is pressed.

property Default: Boolean;

Description If *Default* is *True*, the button's *OnClick* event handler executes when the user presses *Enter*.

Although an application can have more than one Default button, the form calls the *OnClick* event handler only for the first visible button in the tab order. Moreover, any button that has focus becomes the Default button temporarily; hence, if the user selects another button before pressing *Enter*, the selected button's *OnClick* event handler executes instead.

See also *Cancel* property

ModalResult

Determines whether and how the button closes its (modal) parent form.

property ModalResult: TModalResult;

Description Setting the *TButton* component's *ModalResult* property is an easy way to make clicking the button close a modal form. When a button is clicked, the *ModalResult* property of its parent form is set to the same value as the button's *ModalResult* property.

For example, if a dialog box has OK and Cancel buttons, their *ModalResult* properties could be set at design time to *mrOk* and *mrCancel*, respectively. At runtime, clicking the OK button would then change the dialog's *ModalResult* property to *mrOk*, and clicking the Cancel button would change the dialog's *ModalResult* property to *mrCancel*. Unless further processing is required, there is no need to write *OnClick* event handlers for the buttons.

See also *TForm.ModalResult* property

Events

The following table lists the events of *TButton*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TButton:*

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject

Method	Visibility	Documented in
Click	public	TButton
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TButton
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject

Method	Visibility	Documented in
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Click

Click simulates a mouse click, as if the user had clicked the button.

procedure Click; **override**;

Description Calling *Click* assigns the value of the button's *ModalResult* property to the *ModalResult* property of its parent form, and then calls the inherited *Click* method.

See also *ModalResult* property

Create

Creates and initializes a *TButton* instance.

constructor Create(AOwner: TComponent); **override**;

Description Use *Create* to programmatically instantiate a button. *Create*

- Calls the inherited *Create* method.
- Sets the *ControlStyle* of the button to include the values *csSetCaption, csOpaque, csDoubleClicks*.
- Sets *Width* to 75 pixels, and *Height* to 25 pixels.

- Sets the *TabStop* property to *True*.

See also *TWinControl.Create* method

TButtonControl

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TButtonControl

Description

TButtonControl is the base class type for Delphi button objects that are wrappers for the Windows button controls.

TButtonControl encapsulates behavior common to all button controls, check boxes and radio buttons in Delphi.

TButtonControl overrides the *WndProc* method to perform special handling of focus changes for buttons.

See also *TButton* component, *TCustomCheckBox* component, *TRadioButton* component

Properties

The following table lists the properties of *TButtonControl*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl

Property	Visibility	Default	Documented in
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Methods

The following table lists the methods for *TButtonControl*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TWinControl
CreateParented	public	TWinControl

Method	Visibility	Documented in
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl

Method	Visibility	Documented in
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl
WndProc	protected	TButtonControl

WndProc

WndProc overrides the inherited method to handle button messages.

procedure WndProc(**var** Message: TMessage); **override**;

Description *WndProc* enables button objects to receive the input focus when WM_LBUTTONDOWN and WM_LBUTTONDBLCLK messages occur.

See also *TControl.WndProc* method

TBytesField

Unit: DB

TObject • TPersistent • TComponent • TField • TBinaryField • TBytesField

A *TBytesField* object represents a bytes field in a dataset.

Description A bytes field value is a set of unformatted bytes of fixed size.

TBytesField introduces no new properties, events, or methods. It overrides the *DataSize* property and *Create* method to reflect the specifics of a field of type *ftBytes*.

See also *TDataSet object, TVarBytesField* object

Properties

The following table lists the properties of *TBytesField*:

Property	Visibility	Default	Documented in
Alignment	published	taLeftJustify	TField
AsBoolean	public		TField
AsCurrency	public		TField
AsDateTime	public		TField
AsFloat	public		TField
AsInteger	public		TField
AsString	public		TBinaryField
AsVariant	public		TBinaryField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
CurValue	public, read-only		TField
CustomConstraint	published		TField
DataSet	public		TField
DataSize	public, read-only		TBytesField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditMask	public		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField

Property	Visibility	Default	Documented in
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	published	16	TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TBinaryField
VCLComObject	public		TComponent
Visible	published	True	TField

DataSize

DataSize is the amount of memory that is needed to store the value of a Bytes field in its native database format.

```
property DataSize: Word;
```

Description Check *DataSize* to determine the number of bytes required to store a Bytes field component's value. Use *DataSize* to determine the size needed for a buffer when working with the *GetData* and *SetData* methods.

For Bytes fields, *DataSize* is the same as the *Size* property.

See also *GetData* method, *SetData* method, *Size* property

Events

The following table lists the events of *TBytesField*:

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TBytesField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TBytesField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject

Method	Visibility	Documented in
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create creates an instance of *TBytesField*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TBytesField*. After calling the inherited constructor, *Create* sets the *DataType* to *ftBytes*.

It is seldom necessary to explicitly call *Create* because a bytes field object is instantiated automatically for all Bytes fields in a dataset.

See also *DataType* property, *TField.Create* method

TCGIApplication

Unit: CGIApp

TObject • TPersistent • TComponent • TWebApplication • TCGIApplication

TCGIApplication encapsulates a CGI server application.

Description *TCGIApplication* provides the fundamental behavior of a CGI server application. *TCGIApplication* starts running when the CGI server receives an HTTP request message. It creates *TCGIRequest* and *TCGIResponse* objects to represent the request message and any response that should be sent in return, passes these to the dispatcher so that the response can be filled in, and sends the response if it was not already sent by a *TWebActionItem*.

Each CGI server project automatically declares an *Application* variable of type *TCGIApplication* as the instance of the application. This declaration overrides the *Application* variable declared in the Forms unit if the CGIApp unit is declared last in the uses list of the project file. Every CGI server application must include the CGIApp unit.

To work with an ISAPI or NSAPI server, use the *TISAPIApplication* object instead.

See also *TISAPIApplication* object

Properties

The following table lists the properties of *TCGIApplication*:

Property	Visibility	Default	Documented in
ActiveCount	public, read-only		TWebApplication
CacheConnections	public		TWebApplication
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
InactiveCount	public, read-only		TWebApplication
MaxConnections	public		TWebApplication
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
Title	public		TWebApplication
VCLComObject	public		TComponent

Methods

The following table lists the methods of *TCGIApplication*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TWebApplication
CreateForm	public	TWebApplication
DefaultHandler	public	TObject
Destroy	public	TWebApplication
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject

Method	Visibility	Documented in
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
Initialize	public	TWebApplication
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
Run	public	TCGIApplication
SafeCallException	public	TComponent

Run

Run executes the CGI application in response to an HTTP request message.

procedure Run; **override**;

Description When creating a new project, Delphi automatically creates a main program block in the project file that calls the *Run* method. When the application is started in response to an HTTP request message, the CGI application's *Run* method is called.

After calling the inherited *Run* method to set up the default error handler and shut down sequence for the application, *TCGIApplication.Run* creates the appropriate request and response objects for the HTTP request message that started the CGI application. *Run* calls the *HandleRequest* method to pass the request and response on to the dispatcher where the response is filled out and sent. Finally, *Run* frees the request and response objects.

See also *TCGIRequest* object, *TCGIResponse* object, *TWebApplication.HandleRequest* method, *TWebDispatcher* object, *TWinCGIRequest* object, *TWinCGIResponse* object

TCheckBox

Unit: StdCtrls | Palette: Standard

TObject • TPersistent • TComponent • TControl • TWinControl • TButtonControl • TCustomCheckBox • TCheckBox

TCheckBox represents a Windows check box.

Description A *TCheckBox* component presents an option for the user. The user can check the box to select the option, or uncheck it to deselect the option.

See also *TCustomCheckBox* object

Properties

The following table lists the properties of *TCheckBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
Alignment	published	taRightJustify	TCustomCheckBox
AllowGrayed	published	False	TCustomCheckBox
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Caption	published		TControl
Checked	published		TCustomCheckBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl

Property	Visibility	Default	Documented in
Enabled	published	True	TControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	True	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ShowHint	published		TControl
Showing	public, read-only		TWinControl
State	published	cbUnchecked	TCustomCheckBox
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

Events

The following table lists the events of *TCheckBox*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl

Event	Visibility	Documented in
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TCheckBox:*

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomCheckBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject

Method	Visibility	Documented in
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

TCheckListBox

Unit: CheckList | Palette: Additional

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomListBox • TCheckListBox

TCheckListBox displays a scrollable list with check boxes next to each item.

Description *TCheckListBox* is similar to *TListBox*, except that each item has a check box next to it. Users can check or uncheck items in the list.

See also *TCheckBox* object, *TCustomCheckBox* object, *TCustomListBox* object, *TListBox* object

Properties

The following table lists the properties of *TCheckListBox*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
AllowGrayed	published	False	TCheckListBox
BorderStyle	published	bsSingle	TCustomListBox
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomListBox
Checked	public		TCheckListBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
Columns	published	0	TCustomListBox
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent

Property	Visibility	Default	Documented in
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
IntegralHeight	published	False	TCustomListBox
ItemHeight	published		TCustomListBox
ItemIndex	public		TCustomListBox
Items	published		TCustomListBox
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
SelCount	public, read-only		TCustomListBox
Selected	public		TCustomListBox
ShowHint	published		TControl
Showing	public, read-only		TWinControl
Sorted	published	False	TCustomListBox
State	public		TCheckListBox
Style	published	lbStandard	TCustomListBox
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
TabWidth	published	0	TCustomListBox
Tag	published	0	TComponent
Top	published		TControl
TopIndex	public		TCustomListBox
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

AllowGrayed

Determines whether check box can be in a "grayed" state.

property AllowGrayed: Boolean;

Description If *AllowGrayed* is set to *True*, each item has three possible states: checked, unchecked, and grayed. If *AllowGrayed* is set to *False*, each item has only two possible states: checked and unchecked.

See also *Checked* property, *State* property

Checked

Indicates which items are checked.

property Checked[**Index**: Integer]: Boolean;

Description For each member of the *Items* array, *Checked* is *True* if a check mark appears in the item's check box. *Checked* corresponds to the *cbChecked* state; that is, *Checked* is *True* when *State* = *cbChecked* and *False* when *State* = *cbUnchecked* or *State* = *cbGrayed*.

See also *AllowGrayed* property, *Items* property, *Selected* property, *State* property

State

Indicates which items are checked.

property State[**Index**: Integer]: TCheckBoxState;

Description For each member of the *Items* array, *State* indicates whether its check box is selected (*cbChecked*), deselected (*cbUnchecked*), or grayed (*cbGrayed*). The *cbChecked* state corresponds to the Boolean property *Checked*; that is, *State* = *cbChecked* when *Checked* = *True*.

See also *AllowGrayed* property, *Checked* property, *Items* property, *Selected* property, *TCheckBoxState* type, *TCustomCheckBox.State* property

Events

The following table lists the events of *TCheckListBox*:

Event	Visibility	Documented in
OnClick	published	TControl
OnClickCheck	published	TCheckListBox
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnDrawItem	published	TCustomListBox

Event	Visibility	Documented in
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMeasureItem	published	TCustomListBox
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

OnClickCheck

OnClickCheck occurs when the user selects or deselects an item's check box.

property OnClickCheck: TNotifyEvent;

Description Write an *OnClickCheck* event handler to implement any special processing that should occur when the user checks or unchecks an item.

See also *Checked* property

Methods

The following table lists the methods of *TCheckListBox:*

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomListBox
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl

Method	Visibility	Documented in
Create	public	TCustomListBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCheckListBox
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
ItemAtPos	public	TCustomListBox
ItemRect	public	TCustomListBox
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject

Method	Visibility	Documented in
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Destroy

Destroys the *TCheckListBox* instance and frees its memory.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Call *Free* instead. *Free* verifies that the component has not already been freed before it calls *Destroy*.

See also *Free* method

TClientDataSet

Unit: DBClient — Palette: Data Access

TObject • TPersistent • TComponent • TDataSet • TClientDataSet

TClientDataSet is the class that implements a database-independent dataset component that can be used independently in a single-tiered application, or as a client dataset in the client application portion of a multi-tiered database application.

Description *TClientDataSet* is the Delphi object that encapsulates a database-independent, distributed dataset. A client dataset can be used as a

- Simple, fully-functional, stand-alone, flat-file dataset for single-tiered database applications. When used in this manner, an application uses the

client dataset to read from and write to a user's hard disk directly, without accessing a database engine.

- Client dataset in the client application portion of a multi-tiered database application.

Note If a multi-tiered application is designed to support the option of working with data off-line, using the "briefcase" model, a client dataset functions in both of the roles described above. When networked with an application server on site, the client dataset communicates with the application server's data provider, and when a user takes the client application and snapshot of data off-site, the client dataset functions as a flat-file dataset in a single-tiered application.

The client dataset generally communicates with a data provider through an *IProvider* dual interface. When used in the one-tiered and briefcase models, the client dataset's provider interface is automatically generated. When used in a client application as part of a multi-tiered application, the client dataset's provider is usually a *TProvider* object that is part of the application server.

See also *TProvider* object, *TDataSet* object

Properties

The following table lists the properties of *TClientDataSet*:

Property	Visibility	Default	Documented in
Active	published	False	TDataSet
AutoCalcFields	published	True	TDataSet
BOF	public, read-only		TDataSet
Bookmark	public		TDataSet
CanModify	public, read-only		TClientDataSet
ChangeCount	public, read-only		TClientDataSet
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Data	public		TClientDataSet
DataSource	public, read-only		TClientDataSet
DefaultFields	public, read-only		TDataSet
Delta	public, read-only		TClientDataSet
Designer	public, read-only		TDataSet
DesignInfo	public		TComponent
EOF	public, read-only		TDataSet
FetchOnDemand	public	True	TClientDataSet

Property	Visibility	Default	Documented in
FieldCount	public, read-only		TDataSet
FieldDefs	public		TDataSet
Fields	public		TDataSet
FieldValues	public		TDataSet
Filter	published		TClientDataSet
Filtered	published	False	TClientDataSet
FilterOptions	published	[]	TClientDataSet
Found	public, read-only		TDataSet
IndexDefs	public, read-only		TClientDataSet
IndexFieldCount	public, read-only		TClientDataSet
IndexFieldNames	published		TClientDataSet
IndexFields	public		TClientDataSet
IndexName	published		TClientDataSet
KeyExclusive	public		TClientDataSet
KeyFieldCount	public		TClientDataSet
KeySize	public, read-only		TClientDataSet
LogChanges	public		TClientDataSet
MasterFields	published		TClientDataSet
MasterSource	published		TClientDataSet
Modified	public, read-only		TDataSet
Name	published		TDataSet
Owner	public, read-only		TComponent
PacketRecords	published	–1	TClientDataSet
Provider	public		TClientDataSet
ProviderName	published		TClientDataSet
ReadOnly	published	False	TClientDataSet
RecNo	public		TClientDataSet
RecordCount	public, read-only		TClientDataSet
RecordSize	public, read-only		TClientDataSet
RemoteServer	published		TClientDataSet
State	public, read-only		TDataSet
Tag	published	0	TComponent
VCLComObject	public		TComponent

CanModify

Indicates whether or not an application can insert, edit, or delete data in a client dataset.

```
property CanModify: Boolean;
```

Description Check the status of *CanModify* to determine if an application can modify a dataset in any way. If *CanModify* is *True*, the dataset can be modified. If *CanModify* is *False*, the dataset is read-only.

CanModify is set automatically when an application opens a dataset. If the *ReadOnly* property of a client dataset component is *True*, then *CanModify* is set to *False*.

See also *ReadOnly* property

ChangeCount

Indicates the number of changes made to the client dataset.

property ChangeCount: Integer;

Description Check *ChangeCount* to determine how many modifications are pending for the dataset. When updates are applied, and any differences are reconciled by the client dataset, *ChangeCount* is reset.

See also *ApplyUpdates* method, *CancelUpdates* method, *Delta* property, *LogChanges* property, *MergeChangeLog* method, *Reconcile* method, *TProvider* object, *UndoLastChange* method

Data

Receives data packets from a provider.

property Data: OleVariant;

Description Use *Data* to receive and access data packets from a provider. For multi-tiered applications, data packets are typically received from the application server. In one-tiered applications, packets are read from a data file.

The client dataset operates on this local copy of data. For example, the following statement retrieves records from the server and stores them in *Data*:

```
ClientDataSet1.Data := Provider1.Data;
```

See also *Delta* property, *LoadFromFile* method, *Reconcile* method, *TProvider* object

DataSource

Specifies the data source for the client dataset.

property DataSource: TDataSource;

Description *Use DataSource* to examine or set the data source component for a master table used by the dataset.

See also *MasterTable* property, *MasterFields* property, *TDataSource* object

Delta

Returns a packet of change log data from the client to a provider.

property Delta: OleVariant;

Description Use *Delta* to pass a packet of change log data from the client dataset to a provider. *Delta* contains only information about those records inserted, modified, or deleted through the client. *Delta* is passed as an argument to the *ApplyUpdates* and *Reconcile* methods, which use the information in the change log to update the database.

On return from successful application of updates, *Delta* is cleared. Depending on the error tolerance allowed by the client dataset through *ApplyUpdates*, if errors occur, *Delta* contains one of the following:

- All log change data, because the maximum number of tolerated errors was encountered and all changes are rolled back.
- Only those log changes that could not be applied, which indicates that fewer errors occurred than the maximum specified as a tolerance level.

See also *ApplyUpdates* method, *Data* property, *MergeChangeLog* method, *Reconcile* method, *TProvider* object

FetchOnDemand

Indicates whether to fetch data packets from the provider on an as needed basis.

property FetchOnDemand: Boolean;

Description By default *FetchOnDemand* is *True*, meaning that the client dataset fetches additional packets of data as needed (for example, as a user scrolls through data, or conducts a search).

Set *FetchOnDemand* to *False* only if your client application contains code that explicitly fetches more data packets as needed using the *GetNextPacket* method.

See also *GetNextPacket* method

Filter

Specifies the text of the current filter for a dataset.

property Filter: **string**;

Description Use *Filter* to specify a dataset filter. When filtering is applied to a dataset, only those records that meet a filter's conditions are available to an application. *Filter* contains the string that describes the filter condition. For example, the following filter condition displays only those records where the State field is 'CA' or 'MA':

```
State = 'CA' or State = 'MA'
```

To filter strings based on partial comparisons, use an asterisk as a wildcard. For example,

```
State = 'M*'
```

Note Applications can set *Filter* at runtime to change the filtering condition for a dataset at (for example, in response to user input).

See also *Filtered* property, *FilterOptions* property, *FindFirst* method, *FindLast* method, *FindNext* method, *FindPrior* method, *OnFilterRecord* event

Filtered

Specifies whether or not filtering is active for a dataset.

```
property Filtered: Boolean;
```

Description Check *Filtered* to determine whether or not dataset filtering is in effect. If *Filtered* is *True*, then filtering is active. Otherwise *Filtered* is *False*. To apply filter conditions specified in the *Filter* property or the *OnFilterRecord* event handler, set *Filtered* to *True*.

Note When filtering is enabled, user edits to a record may mean that the record no longer meets a filter's test condition. The next time the record is retrieved from the dataset while the filter is in effect, the record may seem to disappear. If that happens, the next record that passes the filter condition becomes the current record.

See also *Filter* property, *FilterOptions* property, *FindFirst* method, *FindLast* method, *FindNext* method, *FindPrior* method, *OnFilterRecord* event

FilterOptions

Specifies whether or not filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

```
type
  TFilterOption = (foCaseInsensitive, foNoPartialCompare);
  TFilterOptions = set of TFilterOption;

property FilterOptions: TFilterOptions
```

Description Set *FilterOptions* to specify whether or not filtering is case insensitive when filtering on string or character fields, and whether or not partial comparisons for matching filter conditions is allowed.

By default, *FilterOptions* is set to an empty set. For filters based on string fields, set *FilterOptions* to *foCaseInsensitive* to catch all variations on a string regardless of capitalization.

To disable partial string matches in the filter expression, include *foNoPartialCompare* in the filter options.

Note To filter strings bases on partial comparisons, use an asterisk as a wildcard. For example,

```
State = 'M*'
```

See also *Filter* property, *Filtered* property, *OnFilterRecord* event, *Locate* method, *Lookup* method

IndexDefs

Contains information about the indexes for a client dataset.

property IndexDefs: TIndexDefs;

Description Examine *IndexDefs* for index information. *IndexDefs* is an array of index items, each of which describes an available index for the dataset. Ordinarily an application accesses or specifies indexes through the *IndexFieldNames* or *IndexFields* properties.

Note The index items in *IndexDefs* may not always reflect the current indexes available for a dataset. Before examining *IndexDefs*, call *IndexDefs.Update* method to refresh the item list.

See also *IndexFieldNames* property, *TIndexDefs* object, *TIndexDef* object

IndexFieldCount

Indicates number of fields that make up the current index for the dataset.

property IndexFieldCount: Integer

Description Check *IndexFieldCount* to determine how many fields are used by the current index for the dataset.

See also *IndexFieldNames* property, *IndexDefs* property

IndexFieldNames

Lists the fields to use as an index for a dataset.

property IndexFieldNames: **string**;

Description Use *IndexFieldNames* as an alternative method of specifying the index to use for a dataset. In *IndexFieldNames* specify the name of each field to use as an index for a dataset. Ordering of field names is significant. Separate names with semicolon.

Tip Use *IndexFieldNames* to create sort orders on the fly at runtime.

Note The *IndexFieldNames* and *IndexName* properties are mutually exclusive. Setting one clears the other.

See also *IndexDefs* property, *IndexName* property

IndexFields

Specifies the array of fields associated with the current index.

```
property IndexFields[Index: Integer]: TField;
```

Description *IndexFields* provides a zero-based array of field objects, each of which corresponds to a field in the current index. Index is an ordinal value indicating the position of a field in the index. The first field in the index is *IndexFields[0]*, the second is *IndexFields[1]*, and so on.

Note Do not set *IndexField* directly. Instead use the *IndexFieldNames* property to order datasets on the fly at runtime.

See also *AddIndex* method, *IndexDefs* property, *IndexFieldNames* property, *IndexName* property

IndexName

Identifies an index for the client dataset.

```
property IndexName: string;
```

Description Use *IndexName* to specify an alternative index for a dataset. Otherwise, if *IndexName* is empty, a dataset's sort order is based on its default ordering in the data packet. Default ordering is determined by the predefined index, DEFAULT_ORDER.

If *IndexName* contains a valid index name, then that index is used to determine sort order of records.

Note *IndexFieldNames* and *IndexName* are mutually exclusive. Setting one clears the other.

See also *IndexFieldNames* property, *IndexFiles* property

KeyExclusive

Specifies the upper and lower boundaries for a range.

```
property KeyExclusive: Boolean;
```

Description Use *KeyExclusive* to specify whether or not a range includes or excludes the records that match the specified starting and ending values for a range. By default, *KeyExclusive* is *True*, meaning that matching values are included.

To restrict a range to those records that are greater than the specified starting value and less than the specified ending value, set *KeyExclusive* to *False*.

See also *FindNearest* method, *GotoNearest* method, *SetRange* method, *SetRangeEnd* method, *SetRangeStart* method

KeyFieldCount

Specifies the number of fields to use when conducting a partial key search on a multi-field key.

```
property KeyFieldCount: Integer;
```

Description Use *KeyFieldCount* to limit a search based on a multi-field key to a consecutive sub-set of those fields. *KeyFieldCount* is zero-based. For example, if the primary key for a dataset consists of three-fields, a partial-key search can be conducted using only the first fields in the key by setting *KeyFieldCount* to *0*.

Note Searches are only conducted based on consecutive key fields beginning with the first field in the key. For example if a key consists of three fields, an application can set *KeyFieldCount* to *0* to search on the first field, *1* to search on the first and second fields, or 2 to search on all fields. By default *KeyFieldCount* is initially set to include all fields in a search.

See also *FindKey* method, *FindNearest* method, *GotoKey* method, *GotoNearest* method, *Locate* method

KeySize

Indicates the size of the current key used for the dataset.

```
property KeySize: Word;
```

Description Examine *KeySize* to determine the size of the current key used for the dataset. *KeySize* corresponds to the size of the key fields in the primary index for the dataset.

See also *RecordSize* property

LogChanges

Indicates whether to log data changes.

```
property LogChanges: Boolean;
```

Description Use *LogChange* to control whether or not changes made to the data in a client dataset are recorded. When *LogChange* is *True* (the default), all changes are recorded and can be returned to the application server. When *LogChange* is *False*, data changes are made in memory, but the changes cannot be passed to the server.

An application that creates a temporary dataset using the *CreateDataSet* method might set *LogChange* to *False* so it can populate the new dataset with an initial set of values, and then set *LogChange* to *True* to resume recording data changes.

Tip When using a client dataset in a one-tiered application where data changes made by the application will never applied to another dataset, you can set *LogChange* to *False* to conserve system resources.

See also *CreateDataSet* method, *MergeChangeLog* method, *UndoLastChange* method

MasterFields

Specifies one or more fields in a master table to link with corresponding fields in this dataset in order to establish a master-detail relationship between them.

property MasterFields: **string**;

Description Use *MasterFields* after setting the *MasterSource* property to specify the names of one or more fields to use in establishing a detail-master relationship between this dataset and the one specified in *MasterSource*.

MasterFields is a string containing one or more field names in the master table. Separate field names with semicolons.

Each time the current record in the master table changes, the new values in those fields are used to select corresponding records in this dataset for display.

Note At design time, use the Field Link designer to establish the master-detail relationship between two datasets.

See also *MasterSource* property

MasterSource

Specifies the name of the data source component for the dataset to use as a master table in establishing a detail-master relationship between this dataset and another one.

property MasterSource: TDataSource;

Description Use *MasterSource* to specify the name of the data source component whose *DataSet* property identifies a dataset to use as a master table in establishing a detail-master relationship between this dataset and another one.

Note At design time choose an available data source from the *MasterSource* property's drop-down list in the Object Inspector.

After setting the *MasterSource* property, specify which fields to use in the master table by setting the *MasterFields* property. At runtime each time the current record in the master table changes, the new values in those fields are used to select corresponding records in this dataset for display.

Note At design time, use the Field Link designer to establish the master-detail relationship between two datasets.

See also *MasterFields* property

PacketRecords

Indicates the number or type of records in a single data packet.

property PacketRecords: Integer;

Description Use *PacketRecords* to specify how many records should be contained in a single packet fetched from the application server, or to indicate that the packet returned should contain only metadata from the server.

When a client dataset is instantiated, *PacketRecords* is automatically set to *–1*, meaning that a single packet should contain all records in the dataset.

If *PacketRecords* is greater than zero, then it specifies the exact number of records to return in a packet.

To retrieve the metadata for a dataset, set *PacketRecords* to *0*. When *PacketRecords* is zero, the application returns all information from the remote database server that define the database's structure, such as table, column, constraint, and domain definitions.

See also *FetchOnDemand* property, *GetNextPacket* method

Provider

Points to the *IProvider* interface used by the client dataset to communicate with a provider object on an application server.

```
property Provider: IProvider;
```

Description Use *Provider* to read or set the *IProvider* interface used by the client dataset to communicate with the *TProvider* object on the application server. Ordinarily an application only queries this property because its value is set automatically based on the *ProviderName* property when a client dataset registers itself with a remote server object in the client application. If an application server has two or more interfaces that are made available to different remote server objects on the client application, the client can switch among them by changing the value of the *Provider* property at runtime.

See also *ProviderName* property, *TProvider* object, *TRemoteServer* object

ProviderName

Specifies the name of the *Provider* property on the application server that is used by this client dataset on the client application.

```
property ProviderName: string;
```

Description Use *ProviderName* to specify the provider object with which to communicate on the server. This property is used to request a specific *IProvider* interface from a remote server object. If the application server contains multiple provider objects, a client application can set *ProviderName* to any of them once the client's remote server object establishes an initial connection to the application server.

See also *Provider* property, *TProvider* object, *TRemoteServer* object

ReadOnly

Specifies whether or not a client dataset is read-only for this application.

property ReadOnly: Boolean;

Description Use the *ReadOnly* property to prevent users from updating, inserting, or deleting data in the dataset. By default, *ReadOnly* is *False*, meaning users can potentially alter a dataset's data.

Note Even if *ReadOnly* is *False*, users may not be able to modify or add data to a dataset.

To guarantee that users cannot modify or add data to a dataset, set *ReadOnly* to *True*.

When *ReadOnly* is *True*, the dataset's *CanModify* property is *False*.

See also *Active* property, *CanModify* property

RecNo

Indicates the current record in the dataset.

property RecNo: Longint;

Description Examine *RecNo* to determine the record number of the current record in the dataset. Applications might use this property with *RecordCount* to iterate through all the records in a dataset, though typically record iteration is handled with calls to *First*, *Last*, *MoveBy*, *Next*, and *Prior*.

See also *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

RecordCount

Indicates the total number of records associated with the dataset.

property RecordCount: Longint;

Description Examine *RecordCount* to determine the total number of records in the dataset. Applications might use this property with *RecNo* to iterate through all the records in a dataset, though typically record iteration is handled with calls to *First*, *Last*, *MoveBy*, and *Prior*.

Note If the *PacketRecords* property is greater than *0*, then the value for *RecordCount* may change as a user scrolls through the data and additional data packets are fetched from the provider.

See also *RecordCount* property, *RecordSize* property, *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

RecordSize

Indicates the size of a record in the dataset.

```
property RecordSize: Word;
```

Description Examine *RecordSize* to determine the physical size, in bytes, of the buffer Delphi allocates to hold a record in the dataset. When a dataset is opened, the *Open* procedure requests record-buffer size information and stores the returned information in *RecordSize*. Delphi uses this information internally. Applications seldom, if ever, require this information.

See also *Open* method

RemoteServer

Specifies the remote server object in the client application that is used by this client dataset for connection purposes.

```
property RemoteServer: TCustomRemoteServer;
```

Description Use *RemoteServer* to specify the remote server object this client dataset should use. The remote server is used to obtain an *IProvider* interface for data exchange in a multi-tiered application. If a client application uses more than one remote server object, the application can change which remote server a client dataset uses by changing the value for this property.

See also *TRemoteServer* object, *ProviderName* property

Events

The following table lists the events of *TClientDataSet*:

Event	Visibility	Documented in
AfterCancel	published	TDataSet
AfterClose	published	TDataSet
AfterDelete	published	TDataSet
AfterEdit	published	TDataSet
AfterInsert	published	TDataSet
AfterOpen	published	TDataSet
AfterPost	published	TDataSet
AfterScroll	published	TDataSet
BeforeCancel	published	TDataSet
BeforeClose	published	TDataSet
BeforeDelete	published	TDataSet
BeforeEdit	published	TDataSet
BeforeInsert	published	TDataSet
BeforeOpen	published	TDataSet
BeforePost	published	TDataSet
BeforeScroll	published	TDataSet
OnCalcFields	published	TDataSet

Event	Visibility	Documented in
OnDeleteError	published	TDataSet
OnEditError	published	TDataSet
OnFilterRecord	published	TDataSet
OnNewRecord	published	TDataSet
OnPostError	published	TDataSet
OnReconcileError	published	TClientDataSet

OnReconcileError

OnReconcileError occurs when a client dataset needs to reconcile an update to a record that could not be resolved by the provider on the application server.

```
type
  TUpdateKind = (ukModify, ukInsert, ukDelete);
  TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel, raRefresh);
  TReconcileErrorEvent = procedure(DataSet: TClientDataSet; E: EReconcileError;
    UpdateKind: TUpdateKind; var Action: TReconcileAction) of object;

property OnReconcileError: TReconcileErrorEvent;
```

Description

Write an *OnReconcileError* event handler to respond to error conditions that arise when the *Reconcile* method cycles through records returned by the application server after it applies updates to a database. Records returned by the server are ones that the server could not apply to the database. *Reconcile* attempts to handle these records. *OnReconcileError* is called once for each record returned by the server.

DataSet is the name of the client dataset to which reconciled records are applied.

E is a pointer to a *EReconcileError* object from which an application can extract an error message and the actual cause of the error condition. An *OnReconcileError* handler can use this information to determine how to respond to the error condition.

UpdateKind describes the type of update that generated the error. The following table lists the possible values for *UpdateKind* and what they indicate:

Value	Meaning
ukModify	Editing an existing record caused an error.
ukInsert	Inserting or appending a new record caused an error.
ukDelete	Deleting an existing record raised an error.

Action indicates the action to take when the *OnReconcileError* handler exits. On entry into the handler, *Action* is always set to the action taken by the resolution process on the server. If *OnReconcileError* can handle or correct the

error, set *Action* to *raRetry* before exiting the error handler. The following table lists the possible values for *Action* and what they indicate:

Value	Meaning
raSkip	Skips updating the record that raised the error condition, and leaves the unapplied changes in the cache.
raAbort	Aborts the operation without displaying an error message.
raMerge	Merges the updated record with the record on the server.
raCorrect	Reapplies the updated record to the database.
raCancel	Removes changes from the change log.
raRefresh	Updates the current record value and removes the changes from the change log.

Note The code in an *OnReconcileError* handler can examine and apply the *OldValue*, *NewValue*, and *CurValue* properties for each field in the current record, but should not call any methods that make a different record the current one.

See also *Reconcile* method

Methods

The following table lists the methods of *TClientDataSet*:

Method	Visibility	Documented in
ActiveBuffer	public	TDataSet
AddIndex	public	TClientDataSet
Append	public	TDataSet
AppendData	public	TClientDataSet
AppendRecord	public	TDataSet
ApplyRange	public	TClientDataSet
ApplyUpdates	public	TClientDataSet
Assign	public	TPersistent
BookmarkValid	public	TClientDataSet
Cancel	public	TDataSet
CancelRange	public	TClientDataSet
CancelUpdates	public	TClientDataSet
CheckBrowseMode	public	TDataSet
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClearFields	public	TDataSet

Method	Visibility	Documented in
CloneCursor	public	TClientDataSet
Close	public	TDataSet
CompareBookmarks	public	TClientDataSet
ControlsDisabled	public	TDataSet
Create	public	TClientDataSet
CreateDataSet	public	TClientDataSet
CursorPosChanged	public	TDataSet
DefaultHandler	public	TObject
Delete	public	TDataSet
DeleteIndex	public	TClientDataSet
Destroy	public	TClientDataSet
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableConstraints	public	TClientDataSet
DisableControls	public	TDataSet
Dispatch	public	TObject
Edit	public	TDataSet
EditKey	public	TClientDataSet
EditRangeEnd	public	TClientDataSet
EditRangeStart	public	TClientDataSet
EnableConstraints	public	TClientDataSet
EnableControls	public	TDataSet
FieldAddress	public	TObject
FieldByName	public	TDataSet
FindComponent	public	TComponent
FindField	public	TDataSet
FindFirst	public	TDataSet
FindKey	public	TClientDataSet
FindLast	public	TDataSet
FindNearest	public	TClientDataSet
FindNext	public	TDataSet
FindPrior	public	TDataSet
First	public	TDataSet
Free	public	TObject
FreeBookmark	public	TDataSet
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetBookmark	public	TDataSet
GetCurrentRecord	public	TClientDataSet
GetFieldList	public	TDataSet
GetFieldNames	public	TDataSet

Method	Visibility	Documented in
GetIndexInfo	public	TClientDataSet
GetIndexNames	public	TClientDataSet
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetNextPacket	public	TClientDataSet
GetParentComponent	public	TComponent
GotoBookmark	public	TDataSet
GotoCurrent	public	TClientDataSet
GotoKey	public	TClientDataSet
GotoNearest	public	TClientDataSet
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
Insert	public	TDataSet
InsertComponent	public	TComponent
InsertRecord	public	TDataSet
InstanceSize	public	TObject
IsEmpty	public	TDataSet
IsLinkedTo	public	TDataSet
Last	public	TDataSet
LoadFromFile	public	TClientDataSet
LoadFromStream	public	TClientDataSet
Locate	public	TClientDataSet
Lookup	public	TClientDataSet
MergeChangeLog	public	TClientDataSet
MethodAddress	public	TObject
MethodName	public	TObject
MoveBy	public	TDataSet
NewInstance	public	TObject
Next	public	TDataSet
Open	public	TDataSet
Post	public	TDataSet
Prior	public	TDataSet
Reconcile	public	TClientDataSet
Refresh	public	TDataSet
RemoveComponent	public	TComponent
Resync	public	TDataSet
RevertRecord	public	TClientDataSet
SafeCallException	public	TComponent
SaveToFile	public	TClientDataSet

Method	Visibility	Documented in
SaveToStream	public	TClientDataSet
SetFields	public	TDataSet
SetKey	public	TClientDataSet
SetRange	public	TClientDataSet
SetRangeEnd	public	TClientDataSet
SetRangeStart	public	TClientDataSet
Translate	public	TDataSet
UndoLastChange	public	TClientDataSet
UpdateCursorPos	public	TDataSet
UpdateRecord	public	TDataSet
UpdateStatus	public	TClientDataSet

AddIndex

Creates a new index for the client dataset.

```
type
  TIndexOptions = set of (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive, ixExpression);

procedure AddIndex(const Name, Fields: string; Options: TIndexOptions);
```

Description Call *AddIndex* to create a new index for the client dataset. *Name* is the name of the new index. *Fields* is a semicolon-delimited list of the fields to include in the index. *Options* is a potentially restricted set of one or more of the following values:

Value	Meaning
ixDescending	Sort in descending alphanumeric order
ixCaseInsensitive	Ignore case when sorting

Attempting to create an index using options that are not applicable to the table raises an exception.

Note Indexes added by *AddIndex* are not saved with data when using *SaveToFile*.

See also *DeleteIndex* method, *IndexDefs* property, *IndexFiles* property, *IndexName* property

AppendData

Adds a newly fetched packet of records to those already stored in the client dataset.

```
procedure AppendData(Data: OleVariant; HitEOF: Boolean);
```

Description Ordinarily, an application does not call *AppendData* directly. Instead an application should call *GetNextPacket* to fetch a packet of records and automatically add them to those already available. *Data* indicates the newly

fetched packet of records to add. *HitEOF* indicates whether or not the end of the dataset was encountered by the application server when it fetched the records from the database server.

See also *GetNextPacket* method, *TProvider* object

ApplyRange

Apply a range to the dataset.

```
procedure ApplyRange;
```

Description Call *ApplyRange* to cause a range established with *SetRangeStart* and *SetRangeEnd*, or *EditRangeStart* and *EditRangeEnd*, to take effect. When a range is in effect, only those records that fall within the range are available to the application for viewing and editing.

See also *CancelRange* method, *EditRangeEnd* method, *EditRangeStart* method, *SetRange* method, *SetRangeEnd* method, *SetRangeStart* method

ApplyUpdates

Sends all updated, inserted, and deleted records from the client dataset to the provider on the application server for writing to the database.

```
function ApplyUpdates(MaxErrors: Integer); Integer;
```

Description Call *ApplyUpdates* to write all updated, inserted, and deleted records from the client dataset to the provider on the application server. *MaxErrors* indicates the maximum number of errors that the provider should allow before prematurely stopping the update operation.

If there are pending updates, *ApplyUpdates* calls the client dataset's *Reconcile* method to send the updates to the server and attempts to reconcile any records returned by the server because they generated errors when the server attempted to apply them to the database.

ApplyUpdates returns the number of errors it encountered. Based on this return value and the setting of *MaxErrors*, successfully applied records are removed from the client dataset's change log. If *MaxErrors* is *–1*, then on error, the update process in aborted, and no records are removed from the change log If *MaxErrors* is a positive number and more errors occur than permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors* all records successfully applied are automatically cleared from the dataset's change log.

See also *Reconcile* method, *TProvider* object

BookmarkValid

Tests the validity of a specified bookmark.

```
type TBookmark = Pointer;
```

```
function BookmarkValid(Bookmark: TBookmark): Boolean;
```

Description Determines if a bookmark is valid. *Bookmark* is the bookmark pointer to test. If the bookmark is valid, *BookmarkValid* returns *True*. Otherwise it returns *False*.

See also *CompareBookmarks* method

CancelRange

Removes any ranges currently in effect for the client dataset.

```
procedure CancelRange;
```

Description Call *CancelRange* to remove a range currently applied to a client dataset. Canceling a range re-enables access to all records in the dataset.

See also *ApplyRange* method, *EditRangeEnd* method, *EditRangeStart* method, *SetRange* method, *SetRangeEnd* method, *SetRangeStart* method

CancelUpdates

Clears all pending edits and clears the change log.

```
procedure CancelUpdates;
```

Description Call *CancelUpdates* to clear all pending updates from the *Delta* property. *Delta* is a variant maintained by the client dataset to record updates, inserts, and deletes.

See also *ApplyUpdates* method, *Delta* property, *RevertRecord* method, *UndoLastChange* method

C

CloneCursor

Points to the data belonging to another client dataset

```
procedure CloneCursor(Source :TClientDataSet; Reset: Boolean);
```

Description Call *CloneCursor* to provide a second editable view of the data belonging to another client dataset. *Source* indicates another client dataset for which to provide a pointer. *Reset* determines whether or not to clone the source dataset's current ranges and filters. If *Reset* is *False* the current range is applied to the cloned dataset. If *Reset* is *True*, the dataset is cloned, but an active range is not applied to the cloned dataset.

The following properties of the specified dataset are cloned:

Property	Property	Property
IndexFieldNames	IndexName	MasterSource
MasterFields	Provider	ProviderName
RemoteServer		

CloneCursor closes the dataset, points to the specified client dataset, and opens a second view on it. For example,

```
ClientDataSet2.CloneCursor(ClientDataSet1, False);
```

After this call, the data for *ClientDataSet2* is the same as the data for *ClientDataSet1*. Edits performed by either client dataset are reflected in the data for *ClientDataSet1*, and are visible from either dataset.

See also *CreateDataSet* method

CompareBookmarks

Compare two bookmarks.

```
type
  TBookmark = Pointer;

function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
```

Description Call *CompareBookmarks* to determine if two bookmarks are identical or not. *Bookmark1* and *Bookmark2* are the bookmarks to compare. *CompareBookmarks* returns *–1* if *Bookmark1* is less than *Bookmark2*, *1* if *Bookmark1* is greater than *Bookmark2*, and *0* if the bookmarks are identical or **nil**.

See also *BookmarkValid* method

Create

Creates an instance of a *TClientDataSet* component.

```
constructor Create(AOwner: TComponent);
```

Description Call *Create* to instantiate a client dataset component at runtime. *Create* clears the *Data* property and sets up the master links for the dataset.

See also *Data* property, *Destroy* method

CreateDataSet

Creates a new, empty client dataset.

```
procedure CreateDataSet;
```

Description Call *CreateDataSet* at runtime to create a new, empty client dataset.

If the *FieldDefs* property contains values, these values are used to create field definitions. Otherwise the *Fields* property is used. One or both of these properties must contain values in order to create a dataset.

If the *IndexDefs* property contain values, these values are used to create index definitions for the dataset.

See also *AddIndex* method, *DeleteIndex* method, *FieldDefs* property, *Fields* property, *IndexDefs* property

DeleteIndex

Deletes an index for the client dataset.

```
procedure DeleteIndex(const Name: string);
```

Description Call *DeleteIndex* to remove an index for a client dataset. *Name* is the name of the index to delete.

Note You cannot delete the reserved indexes, DEFAULT_ORDER and CHANGEINDEX.

See also *AddIndex* method, *IndexDefs* property

Destroy

Destroys the instance of a client dataset component.

```
destructor Destroy;
```

Description Do not call *Destroy* directly in an application. Usually destruction of datasets is handled automatically by Delphi. If an application creates its own instances of a dataset, however, the application should call *Free*, which verifies that the dataset component is not already freed before calling *Destroy*.

Destroy clears the *Data* property, calls its inherited *Destroy* destructor, frees the master links for the client dataset, and frees its index definitions list.

See also *Data* property, *FieldDefs* property, *Create* method

DisableConstraints

Disables application of server constraints on the dataset.

```
procedure DisableConstraints;
```

Description Call *DisableConstraints* to prevent, on a temporary basis, application of server constraints replicated to the dataset. For example, when an application iterates through a dataset, it may improve performance to disable constraints temporarily.

Note Calling *DisableConstraints* increments a reference count. As long as this reference count is greater than zero, constraints are disabled for the dataset. To prevent accidental disabling of constraints, always group a call to *DisableConstraints* with a call to *EnableConstraints*.

See also *EnableConstraints* method

EditKey

Enables modification of the search key buffer.

```
procedure EditKey;
```

Description Call *EditKey* to put the client dataset in *dsSetKey* state while preserving the current contents of the current search key buffer. To determine current search keys, you can use the *IndexFields* property to iterate over the fields used by the current index.

EditKey is especially useful when performing multiple searches where only one or two field values among many change between each search.

See also *Fields* property, *FindKey* method, *GotoKey* method, *GotoNearest* method, *SetKey* method, *State* property

EditRangeEnd

Enables changing the ending value for an existing range.

```
procedure EditRangeEnd;
```

Description Call *EditRangeEnd* to change the ending value for an existing range. To specify an end range value, call *FieldByName* after calling *EditRangeEnd*. After assigning a new ending value, call *ApplyRange* to activate the modified range.

See also *ApplyRange* method, *CancelRange* method, *EditRangeStart* method, *FieldByName* method, *IndexFieldNames* property, *KeyExclusive* property, *SetRange* method, *SetRangeEnd* method, *SetRangeStart* method

EditRangeStart

Enables changing the starting value for an existing range.

```
procedure EditRangeStart;
```

Description Call *EditRangeStart* to change the starting value for an existing range. To specify a start range value, call *FieldByName* after calling *EditRangeStart*. After assigning a new ending value, call *ApplyRange* to activate the modified range.

See also *ApplyRange* method, *CancelRange* method, *EditRangeEnd* method, *FieldByName* method, *IndexFieldNames* property, *SetRange* method, *SetRangeEnd* method, *SetRangeStart* method

EnableConstraints

Enables application of server constraints against dataset entries.

```
procedure EnableConstraints
```

Description Call *EnableConstraints* to turn on application server constraints previously disabled by a call to *DisableConstraints*.

Note Calling *EnableConstraints* decrements a reference count. When this reference count is zero constraints are enabled for the dataset. To prevent accidental

disabling of constraints, always follow a call to *DisableConstraints* with a matching call to *EnableConstraints*.

See also *DisableConstraints* method

FindKey

Searches for a record containing specified field values.

```
function FindKey(const KeyValues: array of const): Boolean;
```

Description Call *FindKey* to search for a specific record in a dataset. *KeyValues* contains a comma-delimited array of field values, called a *key*. Each value in the key can be a literal, a variable, a null, or **nil**. If the number of values passed in *KeyValues* is less than the number of columns in the index used for the search, the missing values are assumed to be null.

If a search is successful, *FindKey* positions the cursor on the matching record and returns *True*. Otherwise the cursor is not moved, and *FindKey* returns *False*.

See also *FindNearest* method, *GotoKey* method, *GotoNearest* method, *IndexName* property, *IndexFieldNames* property, *Locate* method

FindNearest

Moves the cursor to the record that most closely matches a specified set of key values.

```
procedure FindNearest(const KeyValues: array of const);
```

Description Call *FindNearest* to move the cursor to a specific record in a dataset or to the first record in the dataset that is greater than the values specified in the *KeyValues* parameter. *KeyValues* contains a comma-delimited array of field values, called a *key*. If the number of values passed in *KeyValues* is less than the number of columns in the index used for the search, the missing values are assumed to be null.

Note *FindNearest* works only with string data types.

FindNearest positions the cursor either on a record that exactly matches the search criteria, or on the first record whose values are greater than those specified in the search criteria. The current setting of *KeyExclusive* also affects this behavior.

See also *FindKey* method, *GotoKey* method, *GotoNearest* method, *IndexName* property, *IndexFieldNames* property, *KeyExclusive* property, *Locate* method

GetCurrentRecord

Retrieves the current record into a buffer.

```
function GetCurrentRecord(Buffer: PChar): Boolean;
```

Description Most applications should not need to call *GetCurrentRecord*. *TClientDataSet* automatically allocates a buffer for the active record.

Call *GetCurrentRecord* to copy the current record into a buffer allocated by the application. Buffer must be at least as big as the record size indicated by the *RecordSize* property.

See also *RecordSize* property, *ActiveBuffer* method, *TBlobStream* object, *TBlobStream.Create* method

GetIndexInfo

Retrieves information about a dataset's current index.

```
procedure GetIndexInfo;
```

Description Call *GetIndexInfo* to populate a dataset's internal index fields with information about the current index, including its field count, map, and key size.

See also *FieldCount* property, *IndexDefs* property, *KeySize* property, *GetIndexNames* method

GetIndexNames

Retrieves a list of available indexes for a client dataset.

```
procedure GetIndexNames(List: TStrings);
```

Description Call *GetIndexNames* to retrieve a list of all available indexes for a client dataset. *List* is a string list object, created and maintained by the application, into which to retrieve the index names.

Note Two indexes are automatically generated for all client datasets:

- DEFAULT_ORDER, which describes the order of data when it is put into the data packet.
- CHANGEINDEX, which orders the data as it would if it were a delta assigned to another dataset.

See also *IndexName* property

GetNextPacket

Retrieves the next block of records from the application server.

```
function GetNextPacket: Integer;
```

Description Call *GetNextPacket* to retrieve the next sequential block of records from the application server. To specify the maximum number of records to return in a packet, set the *PacketRecords* property before calling *GetNextPacket*. A packet is appended to those records already stored in the *Data* property.

GetNextPacket returns the number of records added to *Data*. If the return value is *0*, the client application has already received all available records from the application server.

See also *AppendData* method, *Data* property, *FetchOnDemand* property, *PacketRecords* property

GotoCurrent

Sets the current record in this dataset the same as the current record in another, cloned dataset.

```
procedure GotoCurrent(DataSet: TClientDataset);
```

Description Call *GotoCurrent* to position the cursor on the same record that is current in the specified dataset. *DataSet* is the cloned dataset whose cursor position in to be matched in this dataset.

GotoCurrent is only useful for synchronizing two client datasets that use separate data sources to view and edit the same data.

See also *CloneCursor* method

GotoKey

Moves the cursor to a record specified by the current key.

```
function GotoKey: Boolean;
```

Description Use *GotoKey* to move to a record specified by key values assigned with previous calls to *SetKey* or *EditKey* and actual search values indicated in the search key buffer.

If *GotoKey* finds a matching record, it positions the cursor on the record and returns *True*. Otherwise the current cursor position remains unchanged, and *GotoKey* returns *False*.

See also *FindKey* method, *FindNearest* method, *GotoNearest* method, *SetKey* method, *State* property

GotoNearest

Moves the cursor to the record that most closely matches the current key.

```
procedure GotoNearest;
```

Description Call *GotoNearest* to position the cursor on the record that is either the exact record specified by the current key values in the key buffer, or on the first record whose values exceed those specified.

Note *KeyExclusive* determines which records are considered part of a search range.

Before calling *GotoNearest*, an application must specify key values by calling *SetKey* or *EditKey* to put the dataset is *dsSetKey* state, and then use *FieldByName* to populate the buffer with search values.

See also *FindNearest* method, *GotoKey* method, *KeyExclusive* property, *SetKey* method, *State* property

LoadFromFile

Loads a client dataset's data from a file.

```
procedure LoadFromFile(const FileName: string);
```

Description Call *LoadFromFile* to populate the *Data* property of the client dataset with data stored in an external file. *FileName* is the name of the file containing the data to load.

Note *FileName* must contain data saved to a file by a previous call to this or another client dataset's *SaveToFile* method.

See also *LoadFromStream* method, *SaveToFile* method

LoadFromStream

Loads a client dataset's data from a stream.

```
procedure LoadFromStream(Stream: TStream);
```

Description Call *LoadFromStream* to populate the *Data* property of the client dataset with data from a stream. *Stream* is the name of the stream from which to read data.

Note *Stream* must contain data in a client dataset's internal block format (for example, data written to a stream by another client dataset using the *SaveToStream* method.

See also *LoadFromFile* method, *SaveToStream* method

Locate

Searches the dataset for a specified record and makes that record the current record.

```
type
  TLocateOption = (loCaseInsensitive, loPartialKey);
  TLocateOptions = set of TLocateOption;

function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean;
```

Description Call *Locate* to search a dataset for a specific record and position the cursor on it. *KeyFields* is a string containing a semicolon-delimited list of field names on which to search.

KeyValues is a variant containing the values to match in the key fields. To specify multiple search values, pass *KeyValues* as a variant array as an argument, or construct a variant array on the fly using the *VarArrayOf* routine. For example,

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver', 'P']), loPartialKey);
```

Options is a set that optionally specifies additional search latitude when searching on string fields. If *Options* contains the *loCaseInsensitive* setting, then *Locate* ignores case when matching fields. If *Options* contains the *loPartialKey* setting, then *Locate* finds the first record that fulfills at least some initial part of the *KeyValues* criteria for record matching. If *Options* is an empty set, or if the *KeyFields* are not string fields, *Options* is ignored.

Locate returns *True* if it finds a matching record, and makes that record the current one. Otherwise *Locate* returns *False*.

See also *FindField* method, *FindFirst* method, *FindKey* method, *Lookup* method

Lookup

Retrieves field values from a record that matches specified search values.

```
function Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant;
```

Description Call *Lookup* to retrieve values for specified fields from a record that matches search criteria. *KeyFields* is a string containing a semicolon-delimited list of field names on which to search.

KeyValues is a variant containing the values to match in the key fields. To specify multiple search values, pass *KeyValues* as a variant array as an argument, or construct a variant array on the fly using the *VarArrayOf* routine.

ResultFields is a string containing a semicolon-delimited list of field names whose values should be returned from the matching record.

Lookup returns a variant array containing the values from the fields specified in *ResultFields*.

See also *Locate* method

MergeChangeLog

Merges any data changes in the change log into the client dataset, overwriting existing values.

```
procedure MergeChangeLog;
```

Description Call *MergeChangeLog* to merge changes in the change log into the actual data for the dataset, effectively creating a new data baseline. Existing values for which changes exist are overwritten.

Note Ordinarily, applications need not call this method. It is called automatically by *ApplyUpdates* as part of the updating process.

See also *Delta* property, *LogChanges* property, *Reconcile* method

Reconcile

Clears successfully updated records from the client dataset's cache of changes.

```
function Reconcile(Results: OleVariant): Boolean;
```

Description *Reconcile* is called by the *ApplyUpdates* function to handle error conditions that occur during the update process. *Reconcile* is called after changes are applied to the dataset managed by a provider from the changes in a client dataset's *Delta* property. (In a multi-tiered application, the provider resides on the application server.)

Results is a variant, declared by an application, that contains error information and record information for each record that could not be applied by the provider.

If all updates are applied successfully, *Results* is null, and all records are automatically cleared from the dataset's change log by a call to *MergeChangeLog*.

If errors occurs during the update process, *Results* contains error records, and records in the change log may not be removed, depending on the setting of the *MaxErrors* parameter in *ApplyUpdates*. If *MaxErrors* is *–1*, then on error, the update process in aborted, and no records are removed from the change log, and the contents of *Results* matches the contents of *Delta*. If *MaxErrors* is a positive number and more errors occur than permitted by *MaxErrors*, all updates are aborted, and *Results* contains only those records that generated an error. If fewer errors occur than specified by *MaxErrors*, *Results* again contains error records, but all records successfully applied are automatically cleared from the dataset's change log.

Reconcile returns a Boolean value indicating whether or not reconciliation is successful. If *True*, Reconcile succeeded without error. If *False*, one or more errors could not be reconciled.

See also *ApplyUpdates* method, *Delta* property, *MergeChangeLog* method, *OnReconcileError* event

RevertRecord

Undoes edits to the current record.

```
procedure RevertRecord;
```

Description Call *RevertRecord* to remove any modifications made to the current record. *RevertRecord* can only remove modifications to records that have not been applied to the database by the application server.

Note To undo the last edit made to a record, call *UndoLastChange*. To cancel all pending modifications, call *CancelUpdates*.

See also *ApplyUpdates* method, *CancelUpdates* method, *Reconcile* method, *UndoLastChange* method

SaveToFile

Saves a client dataset's data to an external file.

```
procedure SaveToFile(const FileName: string);
```

Description Call *SaveToFile* to write a client dataset's data to an external file for later use by this or other client datasets. *FileName* is the name of the external file to use. If the file already exists, its current contents are overwritten.

See also *LoadFromFile* method, *SaveToStream* method

SaveToStream

Saves a client dataset's data to a stream.

```
procedure SaveToStream(Stream: TStream);
```

Description Call *SaveToStream* to write a client dataset's data to a stream for immediate use by other datasets. *Stream* is the name of an existing stream to write to.

See also *LoadFromStream* method, *SaveToFile* method

SetKey

Enables setting of keys and ranges for a dataset prior to a search.

```
procedure SetKey;
```

Description Call *SetKey* to put the dataset into *dsSetKey* state and clear the current contents of the search key buffer. Use *FieldByName* to supply the buffer then with a new set of values prior to conducting a search.

Note To modify an existing key or range, call *EditKey*.

See also *EditKey* property, *FieldByName* method, *GotoKey* method, *SetRange* method, *SetRangeEnd* method, *SetRangeStart* method, *State* property

SetRange

Sets the starting and ending values of a range, and applies it.

```
procedure SetRange(const StartValues, EndValues: array of const);
```

Description Call *SetRange* to specify a range and apply it to the dataset. *StartValues* indicates the field values that designate the first record in the range.

EndValues indicates the field values that designate the last record in the range.

SetRange combines the functionality of *SetRangeStart*, *SetRangeEnd*, and *ApplyRange* in a single procedure call. *SetRange* performs the following functions:

1. Puts the dataset into *dsSetKey* state.
2. Erases any previously specified starting range values and ending range values.
3. Sets the start and end range values.
4. Applies the range to the dataset.

If either *StartValues* or *EndValues* has fewer elements than the number of fields in the current index, then the remaining entries are set to NULL.

See also *ApplyRange* method, *CancelRange* method, *EditRangeEnd* method, *EditRangeStart* method, *FieldValues* property, *IndexFieldNames* property, *KeyExclusive* property, *SetRangeEnd* method, *SetRangeStart* method

SetRangeEnd

Indicates that subsequent assignments to field values specify the end of the range of rows to include in the dataset.

```
procedure SetRangeEnd;
```

Description Call *SetRangeEnd* to put the dataset into *dsSetKey* state, erase any previous end range values, and set them to NULL. Subsequent field assignments made with *FieldByName* specify the actual set of ending values for a range.

After assigning end-range values, call *ApplyRange* to activate the modified range.

See also *ApplyRange* method, *CancelRange* method, *EditRangeEnd* method, *EditRangeStart* method, *FieldByName* method, *IndexFieldNames* property, *KeyExclusive* property, *SetRange* method, *SetRangeStart* method

SetRangeStart

Indicates that subsequent assignments to field values specify the start of the range of rows to include in the dataset.

```
procedure SetRangeStart;
```

Description Call *SetRangeStart* to put the dataset into *dsSetKey* state, erase any previous start range values, and set them to NULL. Subsequent field assignments to *FieldByName* specify the actual set of starting values for a range.

After assigning start-range values, call *ApplyRange* to activate the modified range.

See also *ApplyRange* method, *CancelRange* method, *EditRangeEnd* method, *EditRangeStart* method, *FieldByName* property, *IndexFieldNames* property, *KeyExclusive* property, *SetRange* method, *SetRangeEnd* method

UndoLastChange

Undoes the last edit, insert, or delete operation to a record in the client dataset.

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Description Call *UndoLastChange* to undo the last edit, insert, or delete operation to a record in the client dataset. *FollowChange* indicates whether or not to reposition the cursor on the restored record. If *FollowChange* is *True*, the cursor is positioned on the restored record. If *FollowChange* is *False*, the cursor remains on current record.

See also *LogChanges* property, *ClearLogChanges* method, *CreateDataSet* method

UpdateStatus

Reports the current update status for the current record in the dataset.

```
type
  TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);

function UpdateStatus: TUpdateStatus;
```

Description Call *UpdateStatus* to determine the update status for the current record in the client dataset. Update status can change frequently as records are edited, inserted, or deleted. *UpdateStatus* offers a convenient method for applications to assess the current status before undertaking or completing operations that depend on the update status of the current record.

Note To determine whether or not there are updates pending for an entire dataset, check the *ChangeCount* property.

See also *ApplyUpdates* method, *CancelUpdates* method, *RevertRecord* method

TClientSocket

Unit: ScktComp | Palette: Internet

TObject • TPersistent • TComponent • TCustomSocket • TClientSocket

TClientSocket manages client socket connections for a TCP/IP client.

Description Add a *TClientSocket* object to a form or data module to turn an application into a TCP/IP client. *TClientSocket* specifies a desired connection to a TCP/IP server, manages the connection when it is open, and terminates the connection when the application is through.

See also *TClientWinSocket* object, *TServerSocket* object

Properties

The following table lists the properties of *TClientSocket*:

Property	Visibility	Default	Documented in
Active	published		TCustomSocket
Address	published		TCustomSocket
ClientType	published		TClientSocket
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Host	published		TCustomSocket
Name	published		TComponent
Owner	public, read-only		TComponent
Port	published		TCustomSocket
Service	published		TCustomSocket
Socket	public, read-only		TClientSocket
Tag	published	0	TComponent
VCLComObject	public		TComponent

ClientType

ClientType specifies whether the client socket reads and writes information asynchronously over the socket connection.

```
type TClientType = (ctNonBlocking, ctBlocking);

property ClientType: TClientType;
```

Description Set *ClientType* to *ctNonBlocking* to enable the client socket to respond to asynchronous reading and writing events. When *ClientType* is *ctNonBlocking*, execution is not blocked by reading and writing over the socket connection. *OnRead* or *OnWrite* events occur when the socket needs to read or write over the connection.

Set *ClientType* to *ctBlocking* to force all reading and writing to occur synchronously. It is a good idea to include the client socket object in a thread if the *ClientType* is *ctBlocking*, so that I/O does not block all execution within the client application.

When *ClientType* is *ctBlocking*, use a *TWinSocketStream* object for reading and writing. *TWinSocketStream* prevents the application from hanging indefinitely if a problem occurs while reading or writing. It also can wait for the socket connection to indicate its readiness for reading.

Use a non-blocking socket when the socket needs to synchronize reading and writing with server sockets.

See also *OnRead* event, *OnWrite* event, *TThread* object, *TWinSocketStream* object

Socket

Socket is the *TClientWinSocket* object that describes the client endpoint of the Windows client socket connection.

```
property Socket: TClientWinSocket;
```

Description Use *Socket* to

- Determine the address and port of both the client and server sockets that are bound to the connection.
- Read or write information through the socket connection.
- Determine what server notifications the client will respond to.
- Obtain the Windows socket handle for making Windows socket API calls.

See also *TClientWinSocket* object

Events

The following table lists the events of *TClientSocket*:

Event	Visibility	Documented in
OnConnect	published	TCustomSocket
OnConnecting	published	TCustomSocket
OnDisconnect	published	TCustomSocket
OnError	published	TCustomSocket
OnLookup	published	TCustomSocket
OnRead	published	TCustomSocket
OnWrite	published	TCustomSocket

Methods

The following table lists the methods of *TClientSocket*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject

Method	Visibility	Documented in
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Close	public	TCustomSocket
Create	public	TClientSocket
DefaultHandler	public	TObject
Destroy	public	TClientSocket
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Open	public	TCustomSocket
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Create

Create creates an instance of *TClientSocket*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize a client socket component at runtime. *TClientSocket* objects placed at design time are created automatically.

After calling the inherited constructor, *Create*

- Allocates the *TClientWinSocket* object that describes the client endpoint of the Windows socket connection.
- Initializes the event and error dispatchers for the client socket.

See also *TClientWinSocket* object, *TComponent.Create* method

Destroy

Destroy destroys an instance of *TClientSocket*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the client socket is not already freed, and only then calls *Destroy*.

Destroy frees the *TClientWinSocket* object that describes the client endpoint of the Windows socket connection.

See also *Free* method, *TClientWinSocket* object

TColorDialog

Unit: Dialogs | Palette: Dialogs

TObject • TPersistent • TComponent • TCommonDialog • TColorDialog

TColorDialog displays a modal color-selection dialog.

Description The *TColorDialog* component displays a Windows dialog box for selecting colors. The dialog does not appear at runtime until it is activated by a call to the *Execute* method. When the user selects a color and clicks OK, the dialog closes and the selected color is stored in the *Color* property.

C

Properties

The following table lists the properties of *TColorDialog*:

Property	Visibility	Default	Documented in
Color	published	clBlack	TColorDialog
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Ctl3D	published	False	TCommonDialog

Property	Visibility	Default	Documented in
CustomColors	published		TColorDialog
DesignInfo	public		TComponent
HelpContext	published	0	TCommonDialog
Name	published		TComponent
Options	published	[]	TColorDialog
Owner	public, read-only		TComponent
Tag	published	0	TComponent
VCLComObject	public		TComponent

Color

Returns the selected color.

```
property Color: TColor default clBlack;
```

Description When the user selects a color in the dialog box and clicks OK, the selected color becomes the value of the *Color* property. To make a default color appear in the dialog when it opens, assign a value to *Color* in the Object Inspector or in program code.

See also *TColor* type

CustomColors

Determines which custom colors are available in the dialog box.

```
property CustomColors: TStrings;
```

Description Each custom color is represented as a string of the form *ColorX=HexValue*. For example, the following string sets the first custom color.

```
ColorA=808022
```

Up to 16 custom colors (*ColorA* through *ColorP*) can be set.

Options

Specifies options and default appearance for the dialog.

```
TColorDialogOption = (cdFullOpen, cdPreventFullOpen, cdShowHelp, cdSolidColor, cdAnyColor);
TColorDialogOptions = set of TColorDialogOption;

property Options: TColorDialogOptions default [];
```

Description Use the *Options* property to customize the appearance and functionality of the dialog. The possible values of *Options* are

Value	Meaning
cdFullOpen	Displays the custom color options when the dialog opens.
cdPreventFullOpen	Disables the Define Custom Colors button in the dialog, so that the user cannot define new colors.
cdShowHelp	Adds a Help button to the dialog.
cdSolidColor	Directs Windows to use the nearest solid color to the color chosen.
cdAnyColor	Allows the user to select non-solid colors (which may have to be approximated by dithering).

By default, all of these options are off.

Events

The following table lists the events of *TColorDialog*:

Event	Visibility	Documented in
OnClose	published	TCommonDialog
OnShow	published	TCommonDialog

Methods

The following table lists the methods of *TColorDialog*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TColorDialog
DefaultHandler	public	TObject
Destroy	public	TColorDialog
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
Execute	public	TColorDialog
FieldAddress	public	TObject
FindComponent	public	TComponent

Method	Visibility	Documented in
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Create

Creates and initializes a color-selection dialog.

constructor Create(AOwner: TComponent); **override**;

Description Create generates a *TColorDialog* instance and initializes the *CustomColors* property.

See also *CustomColors* property, *TCommonDialog.Create* method

Destroy

Destroys the color-selection dialog and frees its memory.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Call *Free* instead. *Free* verifies that the color-selection dialog is not already destroyed before calling *Destroy*.

See also *Free* method

Execute

Displays the color-selection dialog.

function Execute: Boolean; **override**;

Description *Execute* opens the color-selection dialog, returning *True* when the user selects a color and clicks OK.

TComboBox

Unit: StdCtrls | Palette: Standard

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomComboBox • TComboBox

TComboBox combines an edit box with a scrollable list.

Description A *TComboBox* component is an edit box with a scrollable drop-down list attached to it. Users can select an item from the list or type directly into the edit box.

See also *TCustomComboBox* object, *TEdit* object, *TListBox* object

Properties

The following table lists the properties of *TComboBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomComboBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent

Property	Visibility	Default	Documented in
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
DropDownCount	published	8	TCustomComboBox
DroppedDown	public		TCustomComboBox
Enabled	published	True	TControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
ItemHeight	published		TCustomComboBox
ItemIndex	public		TCustomComboBox
Items	published		TCustomComboBox
Left	published		TControl
MaxLength	published	0	TCustomComboBox
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
SelLength	public		TCustomComboBox
SelStart	public		TCustomComboBox
SelText	public		TCustomComboBox
ShowHint	published		TControl
Showing	public, read-only		TWinControl
Sorted	published	False	TCustomComboBox
Style	published	csDropDown	TCustomComboBox
TabOrder	published	-1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	published		TControl
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

Events

The following table lists the events of *TComboBox*:

Event	Visibility	Documented in
OnChange	published	TCustomComboBox
OnClick	published	TControl
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnDrawItem	published	TCustomComboBox
OnDropDown	published	TCustomComboBox
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMeasureItem	published	TCustomComboBox
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TComboBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomComboBox
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomComboBox
CreateParented	public	TWinControl

Method	Visibility	Documented in
DefaultHandler	public	TObject
Destroy	public	TCustomComboBox
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl

Method	Visibility	Documented in
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectAll	public	TCustomComboBox
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

TCommonDialog

Unit: Dialogs

TObject • TPersistent • TComponent • TCommonDialog

TCommonDialog is the ancestor of all components that represent Windows common dialog boxes.

Description

TCommonDialog is an abstract class that represents Windows common dialog boxes. Its descendants handle such tasks as finding and opening files, setting fonts and colors, and printing. The dialogs do not appear at runtime until activated by a call to the *Execute* method. Except for *TFindDialog* and *TReplaceDialog*, these dialogs are all modal.

The *TCommonDialog* descendants are nonvisual components that generate a single function call to COMMDLG.DLL, and are therefore less easily modified than other Delphi components. New components that encapsulate common-dialog functions should descend from *TCommonDialog*. Most other dialog boxes should descend from *TForm*.

See also

TColorDialog object, *TFindDialog* object, *TFontDialog* object, *TForm* object, *TOpenDialog* object, *TOpenPictureDialog* object, *TPrintDialog* object, *TPrinterSetupDialog* object, *TReplaceDialog* object, *TSaveDialog* object, *TSavePictureDialog* object

Properties

The following table lists the properties of *TCommonDialog*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Ctl3D	published	True	TCommonDialog
DesignInfo	public		TComponent
Handle	public, read-only		TCommonDialog
HelpContext	published	0	TCommonDialog
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
Template	protected		TCommonDialog
VCLComObject	public		TComponent

Ctl3D

Determines whether the dialog has a three-dimensional look.

property Ctl3D: Boolean;

Description If *Ctl3D* is set to *True*, the dialog box and the controls in it appear three-dimensional. If *Ctl3D* is set to *False*, the dialog and its controls appear flat.

Note For dialogs to appear three-dimensional under Windows NT 3.51, CTL3D32.DLL must be installed in the System32 directory.

See also *TWinControl.Ctl3D* property

Handle

The window handle of the dialog.

property Handle: HWnd;

Description The *Handle* property, which provides access to the dialog's window handle after a call to *Execute*, can be used in *OnShow* and *OnClose* event handlers. *Handle* is reset to 0 after the dialog closes.

See also *Execute* method, *OnClose* event, *OnShow* event

HelpContext

A context number for online Help.

property HelpContext: THelpContext;

Description The *HelpContext* property is an integer value that determines which Help screen appears when the user requests context-sensitive online Help. If *HelpContext* is set to the default value of 0, the dialog inherits the Help context of its parent.

See also *THelpContext* type, *TWinControl.HelpContext* property

Template

Points to a custom template for the dialog.

property Template: PChar;

Description *Template* points to a resource identifier (a null-terminated string or an integer created with *MAKEINTRESOURCE*) that identifies a Windows dialog resource in the current *HInstance* module. The *Execute* method uses this resource to customize the appearance of the dialog.

For example, if 123 is an integer identifier in the resource file MYDLG.RES, the following code could be used customize *OpenDialog1*.

```
{$R Mydlg.res}
OpenDialog1.Template := MakeIntResource(123);
```

If *DialogA* is a string identifier in MYDLG.RES, then the following code could be used customize *PrintDialog1*.

```
{$R Mydlg.res}
PrintDialog1.Template := 'DialogA';
```

See also *Execute* method, *HInstance* variable, *TOpenDialog* object, *TPrintDialog* object

C

Events

The following table lists the events of *TCommonDialog*:

Event	Visibility	Documented in
OnClose	published	TCommonDialog
OnShow	published	TCommonDialog

OnClose

OnClose occurs when the dialog closes.

property OnClose: TNotifyEvent;

Description Write an *OnClose* event handler to perform special processing when the dialog closes.

See also *TNotifyEvent* type, *Handle* property

OnShow

OnShow occurs when the dialog opens.

```
property OnShow: TNotifyEvent;
```

Description Write an *OnShow* event handler to perform special processing when the dialog is displayed.

See also *TNotifyEvent* type, *Handle* property, *Execute* method

Methods

The following table lists the methods of *TCommonDialog*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TCommonDialog
Destroy	public	TCommonDialog
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
Execute	protected	TCommonDialog
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent

Method	Visibility	Documented in
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Create

Creates and initializes a *TCommonDialog* instance.

constructor Create(AOwner: TComponent); **override**;

Description *Create* calls the inherited constructor method and initializes the *Ctl3D* property to *True*.

See also *Ctl3D* property, *TComponent.Create* method

Destroy

Destroys the *TCommonDialog* instance and frees its memory.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Call *Free* instead. *Free* verifies that the dialog is not already destroyed before calling *Destroy*.

See also *Free* method

Execute

Displays the dialog box.

function Execute: Boolean; **virtual**; **abstract**;

Description *TCommonDialog* introduces the abstract *Execute* method, which descendent classes use to open the dialog instance at runtime. Generally, *Execute* returns *True* when the user makes a selection and clicks OK, and returns *False* when the user closes the dialog without making a selection.

TComponent

Unit: Classes

TObject • TPersistent • TComponent

TComponent is the common ancestor of all component objects in Delphi.

Description *TComponent* encapsulates the fundamental behavior common to all components in Delphi by introducing methods and properties that provide:

- The ability to appear on Component palette and be manipulated in the form designer.
- The ability to own and manage other components.
- Enhanced streaming and filing capabilities.
- The ability for components to be used as wrappers for ActiveX controls and other objects that implement interfaces.

Do not create instances of *TComponent*. Use *TComponent* as a base class when declaring non-visual components that can appear on the component palette and be used in the form designer. Properties and methods of *TComponent* provide basic behavior that descendent classes inherit as well as behavior that components can override to customize their behavior.

To create visual components, use *TControl* or its descendents as a base. To create visual windowed controls use *TWinControl* or its descendents as a base.

See also *TControl* object, *TFiler* object, *TStream* object, *TWinControl* object

Properties

The following table lists the properties of *TComponent*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
VCLComObject	public		TComponent

ComObject

ComObject returns the interface reference implemented by components that support COM.

property ComObject: IUnknown;

Description Use *ComObject* to assign the interface implemented by a component to an interface reference. This property is used by VCL components that support COM interfaces. If the component is not a wrapper for a COM component, *ComObject* raises an *EComponentError* exception.

See also *EComponentError* exception

ComponentCount

ComponentCount indicates the number of components owned by the component as listed in the *Components* array property list.

property ComponentCount: Integer;

Description Use *ComponentCount* to find or verify the number of components owned by a component, or when iterating through the *Components* list to perform some action on all owned components. *ComponentCount* is used internally in Delphi for such iterative procedures.

Note The *ComponentCount* of a component contains the same number of items as in the *Components* list for that component, and is always 1 more than the highest *Components* index, because the first *Components* index is always 0.

See also *ComponentIndex* property, *Components* property

ComponentIndex

ComponentIndex indicates the position of the component in its owner's *Components* array property list.

property ComponentIndex: Integer;

Description Use *ComponentIndex* when iterating through the *Components* list to perform some action on owned components. For convenience it can be used in conjunction with *ComponentCount*. *ComponentIndex* is used internally in Delphi for interactive assignment procedures.

Note The first component in the list has a *ComponentIndex* value of 0, the second has a value of 1, and so on. Therefore, when using *ComponentIndex* with *ComponentCount*, note that *ComponentCount* is always 1 more than the highest *Components* index.

See also *ComponentCount* property, *Components* property

Components

Components lists of all components owned by the component.

```
property Components[Index: Integer]: TComponent;
```

Description Use *Components* to access any of these owned components, such as the controls owned by a form. The *Components* property is most useful when needing to refer to owned components by number rather than name. It is also used internally in Delphi for iterative processing on all owned components.

Note For convenience use *Components* with *ComponentCount* for iterative processing. However, be aware that while the *ComponentCount* of a component contains the same number of items as in the *Components* list for that component, *ComponentCount* is always 1 more than the highest *Components* index, because the first *Components* index is always 0.

See also *ComponentCount* property, *ComponentIndex* property, *Owner* property

ComponentState

ComponentState describes the current state of the component, indicating when a component needs to avoid certain actions.

```
type TComponentState = set of (csLoading, csReading, csWriting, csDestroying, csDesigning,
  csAncestor, csUpdating, csFixups);

property ComponentState: TComponentState;
```

Description Components use the *ComponentState* property to detect states in which certain kinds of actions are allowed or disallowed. For example, if a component needs to avoid certain behaviors at design time that it will perform at runtime, it can check for the *csDesigning* flag. *ComponentState* is read-only and its flags are set automatically by various methods, each for its own purpose.

The *TComponentState* type defines the set of possible state flags for the *ComponentState* property. The following table lists the possible values for the *TComponentState* type and the meaning corresponding to each flag:

Flag	Component state
csAncestor	Set if the component was introduced in an ancestor form. Only set if csDesigning is also set.
csDesigning	Design mode, meaning it is in a form being manipulated by a form designer.
csDestroying	The component is about to be destroyed.
csFixups	Set if the component is linked to a component in another form that has not yet been loaded. This flag is cleared when all pending fixups are resolved.
csLoading	Loading from a filer object.
csReading	Reading its property values from a stream.

Flag	Component state
csUpdating	The component is being updated to reflect changes in an ancestor form. Only set if csAncestor is also set.
csWriting	Writing its property values to a stream.

See also *Destroying* method, *Loaded* method, *ReadState* method, *SetDesigning* method, *SetAncestor* method, *Updated* method, *Updating* method, *WriteState* method

ComponentStyle

ComponentStyle sets a a flag for a style that governs the behavior of the component.

```
type TComponentStyle = set of (csInheritable, csCheckPropAvail);
```

```
property ComponentStyle: TComponentStyle;
```

Description *ComponentStyle* determines whether a component (and therefore the forms that contain it) is inheritable, and whether it needs to check its properties for readability.

The *TComponentStyle* type defines a set of values for the *ComponentStyle* property. *TComponent* sets the *csInheritable* style by default in the constructor, and most of the standard components are inheritable.

The *csInheritable* style indicates that the component can be inherited by a descendant form type. If any of the components in a form do not have the *csInheritable* style, the form cannot be used as the ancestor of an inherited form.

The *csCheckPropAvail* style indicates that a component needs to check its properties for readability. This is only used for OLE controls, where the Object Inspector cannot tell directly that a property is readable, and therefore displayable. Native Delphi components should not use this style.

C

DesignInfo

DesignInfo contains information used by the Form designer.

```
property DesignInfo: Longint;
```

Description *DesignInfo* is used internally by the Delphi environment. Do not use this property in applications.

See also *TFormDesigner* object

Name

Name contains the name of the component as referenced by other components.

```
type TComponentName: string;
```

property Name: TComponentName;

Description Use *Name* to change the name of a component to reflect its purpose in the current application. By default, Delphi assigns sequential names based on the type of the component, such as 'Button1', 'Button2', and so on.

See also *ChangeName* method, *SetName* method, *ValidateRename* method

Owner

Owner indicates which component owns the component.

property Owner: TComponent;

Description Use *Owner* to find the owner of a component. When one component is owned by another, the memory for the owned component is freed when its owner's memory is freed. This means that when a form is destroyed, all the components on the form are also destroyed.

In Delphi, a form owns all components that are on it. In turn, the form is owned by the application. Thus when the memory for the application itself is freed, the memory for the form (and all its owned components) is also freed.

The owner of a component is determined by the parameter passed to the *Create* method when the component is constructed. For components created in the Delphi form designer, the owner is passed automatically to the constructor.

See also *Components* property, *Create* method, *Destroy* method, *Free* method

Tag

Tag stores an integer value as part of a component.

property Tag: Longint;

Description *Tag* has no predefined meaning to Delphi. The *Tag* property is provided for the convenience of storing additional integer value or pointer information for special needs in an application. For example, use the *Tag* property when implementing case statements with a component.

VCLComObject

VCLComObject is used internally by Delphi for components that support COM.

property VCLComObject: Pointer;

Description *VCLComObject* is for internal use only. To access the interfaces implemented by a VCL COM component, use the *ComObject* property.

See also *ComObject* property

Methods

The following table lists the methods of *TComponent*:

Method	Visibility	Documented in
Assign	public	TPersistent
ChangeName	protected	TComponent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TComponent
DefaultHandler	public	TObject
DefineProperties	protected	TComponent
Destroy	public	TComponent
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetChildOwner	protected	TComponent
GetChildParent	protected	TComponent
GetChildren	protected	TComponent
GetIDsOfNames	protected	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	protected	TComponent
GetOwner	protected	TComponent
GetParentComponent	public	TComponent
GetTypeInfo	protected	TComponent
GetTypeInfoCount	protected	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject

Method	Visibility	Documented in
Invoke	protected	TComponent
Loaded	protected	TComponent
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TComponent
QueryInterface	protected	TComponent
ReadState	protected	TComponent
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetAncestor	protected	TComponent
SetChildOrder	protected	TComponent
SetDesigning	protected	TComponent
SetName	protected	TComponent
SetParentComponent	protected	TComponent
Updated	protected	TComponent
UpdateRegistry	protected	TComponent
Updating	protected	TComponent
ValidateContainer	protected	TComponent
ValidateInsert	protected	TComponent
ValidateRename	protected	TComponent
WriteState	protected	TComponent
_AddRef	protected	TComponent
_Release	protected	TComponent

ChangeName

ChangeName sets the private, internal storage for the *Name* property to the string passed in *NewName*.

```
type TComponentName: string;

procedure ChangeName(const NewName: TComponentName);
```

Description *ChangeName* is used by the *SetName* method to actually change the component's name. *ChangeName* is not virtual; do not override it. When overriding *SetName* be sure to call the inherited *SetName*.

See also *Name* property, *SetName* method, *ValidateRename* method

Create

Create automatically allocates enough memory and constructs a safely initialized instance of a component.

```
constructor Create(AOwner: TComponent);
```

Description All objects in Delphi have a *Create* method that constructs the object. *TComponent* redefines *Create* so that, for components, *Create* also:

- establishes the relationship of a component and its *Owner*, as indicated by the AOwner parameter
- sets the *ComponentStyle* property to *csInheritable*, meaning that the component can be inherited by a descendant form type

It is not necessary to call *Create* manually for components created in the Delphi form designer. These components are created automatically when the application is run, and they are destroyed when the application is closed.

For components created manually, that is, not created in the form designer, call *Create* and pass in an owner component as the AOwner parameter. The owner will dispose of the component when the owner is destroyed. If the component is not owned, then call *Free* so that it is destroyed.

Note When passing in *Self*, consider what *Self* references. If a component creates another component in one of its methods, then *Self* refers to the first component and not the component being created, which will be owned by the first component.

Note *TComponent.Create* is virtual in part to allow polymorphic instantiation of class references. This is critical to the streaming system and to the form designer. Do not forget to use the override directive when declaring a new component's *Create* constructor.

See also *Components* property, *ComponentStyle* property, *TObject.Create* method, *Destroy* method, *TObject.Free* method, *Owner* property

DefineProperties

DefineProperties designates methods for storing an object's unpublished data on a stream such as a form file.

```
procedure DefineProperties(Filer: TFiler); override;
```

Description *TComponent* overrides the *DefineProperties* method defined in *TPersistent* to define "fake" Top and Left properties. These are defined so that a non-visual component can be manipulated at design-time. However, the Top and Left properties are hidden, that is, they are not published because non-visual components do not appear at runtime.

DefineProperties is virtual, so descendent classes can override it. When writing or overriding a *DefineProperties* method, be aware that the *Ancestor* property of *TFiler* might be set, and that it might be necessary to write or not write properties, as appropriate.

DefineProperties is called automatically as part of the streaming system; do not call it directly.

See also *TFiler.Ancestor* property, *TFiler.DefineBinaryProperty* method, *TPersistent.DefineProperties* method, *TFiler.DefineProperty* method

Destroy

Destroy disposes of the component and its owned components.

destructor Destroy; **override**;

Description *TComponent* overrides *Destroy* to dispose of the component where the method is executed as well as any components that it owns.

Note It is not advisable to call *Destroy* directly. Call *Free* instead.

Calling *Free* for a component, will call *Free* for all components that it owns, that is, all components in its component list. A form owns all the controls and non-visual components that are created on it in design mode. When it is freed, all of these components are automatically freed as well. Since, by default, all forms are owned by the *Application* object, when the application terminates, it frees the *Application* object, which frees all forms. For all objects that are not components, or for components created with a **nil** owner, be sure to call *Free* after finishing with the object; otherwise the memory allocated to the object will be lost until the application terminates.

Never explicitly free a component within one of its own event handlers, nor free a component from an event handler of a component that the component owns or contains.

To free the form, call the *Release* method, which destroys the form and releases the memory allocated for it after all its event handlers and those of the components it contains are through executing.

See also *TObject.Destroy* method, *DestroyComponents* method, *Destroying* method, *Free* method

DestroyComponents

DestroyComponents iterates through the components owned by the component, removing each from the list of owned components and destroying it.

procedure DestroyComponents;

Description *DestroyComponents* ensures that owned components are automatically destroyed by their owner.

It is not necessary to call *DestroyComponents* directly. *DestroyComponents* is automatically called by *Destroy* in the destructor sequence that begins with a call to *Free* and ends with the object destroying itself.

See also *Destroy* method, *Destroying* method, *Free* method, *Owner* method

Destroying

Destroying sets the *csDestroying* flag for the component and all owned components, indicating that the components are about to be destroyed.

```
procedure Destroying;
```

Description *Destroying* sets the *csDestroying* flag in the *ComponentState* property, then calls the *Destroying* method for each owned component so that their *csDestroying* flags will also be set. If *csDestroying* is already set, *Destroying* does nothing.

It is not necessary to call *Destroying* directly. *Destroying* is automatically called by *Destroy* in the destructor sequence that begins with a call to *Free* and ends with the object destroying itself.

See also *ComponentState* property, *Destroy* method, *DestroyComponents* method, *Free* method, *Owner* method

FindComponent

FindComponent indicates whether or not a given component, *AName*, is owned by the component.

```
function FindComponent(const AName: string): TComponent;
```

Description *FindComponent* returns the component in the *Components* array property with the name that matches the string in the *AName* parameter. Use to find whether a given component is owned by another.

FindComponent is not case sensitive.

See also *Components* property, *Owner* property

FreeNotification

FreeNotification ensures that *AComponent* is notified that the component is going to be destroyed.

```
procedure FreeNotification(AComponent: TComponent);
```

Description Use *FreeNotification* when a component is assigned to a property of another component.

It is necessary to call *FreeNotification* for components in other forms that have references to the component. *But* for components in the same form with the component, the *Notification* method will be called automatically.

See also *Notification* method

FreeOnRelease

FreeOnRelease frees the interface reference for components that were created from COM classes.

```
procedure FreeOnRelease;
```

Description *FreeOnRelease* is called when an interface implemented by the component is released. *FreeOnRelease* is used internally and calls the corresponding interface method. It should not be necessary to call *FreeOnRelease* directly.

GetChildOwner

GetChildOwner returns the owner of a child component being read from a stream.

```
function GetChildOwner: TComponent; dynamic;
```

Description In *TComponent GetChildOwner* returns **nil** by default, meaning that the owner is assumed to be the root component currently being read (that is usually a form). The *Owner* of a component is responsible for destroying it.

GetChildOwner is used internally by the streaming system in Delphi. It is rarely necessary to call it directly.

See also *Create* method, *Owner* property

GetChildParent

GetChildParent returns the parent, or if there is no parent, returns the owner of a child component being read from a stream.

```
function GetChildParent: TComponent; dynamic;
```

Description By default *GetChildParent* for *TComponent* returns *Self*. If *GetChildParent* returns **nil**, the parent is assumed to be the root component currently being read (that is usually a form).

GetChildParent is used internally in the Delphi streaming system. It is not necessary to call it directly.

See also *GetParentComponent* method

GetChildren

The *GetChildren* method defined by *TComponent* does nothing other than provide an interface for a method that windowed controls use to return their child components.

```
procedure GetChildren(Proc: TGetChildProc; Root: TComponent); dynamic;
```

Description The *GetChildren* method is introduced in *TComponent* for convenience for the Delphi streaming system. Descendent classes override *GetChildren* to return the "child" components of the component; that is, those who return the component from their *GetParentComponent* method.

See also *GetParentComponent* method, *SetChildOrder* method, *TWriter.WriteData* method

GetIDsOfNames

GetIDsOfNames maps a single member and an optional set of argument names to a corresponding set of integer dispatch identifiers (dispIDs), which can be used on subsequent calls to the *Invoke* method.

```
function GetIDsOfNames(const IID: TGUID; Names: Pointer; NameCount, LocaleID: Integer;
  DispIDs: Pointer): Integer; stdcall;
```

Description *GetIDsOfNames* is the implementation of the *IDispatch* interface *GetIDsOfNames* method. For components that support interfaces, *GetIDsOfNames* calls the implementation of this method for the interface supported by the component, passing the specified parameters.

See also *GetTypeInfo* method, *GetTypeInfoCount* method, *Invoke* method

GetNamePath

GetNamePath returns a string used by the Object Inspector.

```
function GetNamePath: string; override;
```

Description *GetNamePath* is for internal use only. It is used to determine the text to display in the Object Inspector for the name of the object being edited. *GetNamePath* is introduced in *TPersistent* so that descendents like collections can appear in the Object Inspector. *TComponent* overrides *GetNamePath* to return the component name. Do not call *GetNamePath* directly.

See also *GetOwner* method, *TPersistent.GetNamePath* method

GetOwner

GetOwner returns the owner of a component.

```
function GetOwner: TPersistent; override;
```

Description *GetOwner* is used by the *GetNamePath* method to find the owner of a component. *GetNamePath* and *GetOwner* are introduced in *TPersistent* so that descendents like collections can appear in the Object Inspector. For *TComponent* it is the value of the *Owner* property.

See also *GetNamePath* method, *Owner* property, *TPersistent.GetOwner* method

GetParentComponent

GetParentComponent returns **nil** for *TComponent*.

```
function GetParentComponent: TComponent; dynamic;
```

Description The *GetParentComponent* method is introduced in *TComponent* for convenience for the Delphi streaming system. Descendent classes can override *GetParentComponent* to return the appropriate parent for their particular type of component.

See also *TControl.GetParentComponent* method, *HasParent* method, *SetParentComponent* method

GetTypeInfo

GetTypeInfo retrieves the type information for an object, which can be used to get the type information for an interface implemented by the component.

```
function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): Integer; stdcall;
```

Description *GetTypeInfo* is the implementation of the *IDispatch* interface *GetTypeInfo* method. For components that support interfaces, *GetTypeInfo* calls the implementation of this method for the interface supported by the component, passing it the specified parameters.

See also *GetIDsOfNames* method, *GetTypeInfoCount* method, *Invoke* method

GetTypeInfoCount

GetTypeInfoCount retrieves the number of type information interfaces that an object provides (either 0 or 1).

```
function GetTypeInfoCount(out Count: Integer): Integer; stdcall;
```

Description *GetTypeInfoCount* is the implementation of the *IDispatch* interface *GetTypeInfoCount* method. For components that support interfaces, *GetTypeInfoCount* calls the implementation of this method for the interface supported by the component. The **out** parameter points to a location that receives the number of type information interfaces provided by the object. If the object provides type information, this number is 1; otherwise the number is 0.

See also *GetIDsOfNames* method, *GetTypeInfo* method, *Invoke* method

HasParent

HasParent returns *False* for *TComponent*.

```
function HasParent: Boolean; dynamic;
```

Description The *HasParent* method is introduced in *TComponent* for the streaming system to determine whether the component has a parent to handle its filing for it.

A return value of *True* indicates that some other object (a parent) is responsible for writing the object to a stream. Most commonly the parent other object is a form or panel component that contains a control.

It is rarely necessary to override *HasParent*. Any child component that returns *True* from *HasParent* must now also implement the *GetParentComponent* and *SetParentComponent* methods.

See also *TControl.HasParent* method, *TWinControl.GetChildren* method, *GetParentComponent* method, *SetParentComponent* method

InsertComponent

InsertComponent establishes the component as the owner of the component passed in the *AComponent* parameter.

```
procedure InsertComponent(AComponent: TComponent);
```

Description *InsertComponent* adds the component to the end of the *Components* array property. The inserted component must have no name (no specified *Name* property value), or the name must be unique among all others in the *Components* list.

When the owning component is destroyed, AComponent is destroyed also.

Components are automatically inserted and removed when visually manipulating them in the Delphi form designer. Use *InsertComponent* when manually adding components to another *Owner* component's *Components* list.

See also *Components* property, *Owner* property

Invoke

Invoke provides access to OLE Automation properties and methods exposed by an object that supports the *IDispatch* interface.

```
function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer; Flags: Word; var Params;
    VarResult, ExcepInfo, ArgErr: Pointer): Integer; stdcall;
```

Description *Invoke* is the implementation of the *IDispatch* interface *Invoke* method. *Invoke* is a standard mechanism for accessing the exposed properties and methods of an OLE Automation object. For components that support interfaces, *Invoke* calls the implementation of this method for the interface supported by the component, passing it the parameters specified by the function.

See also *GetIDsOfNames* method, *GetTypeInfo* method, *GetTypeInfoCount* method

Loaded

Loaded clears the *csLoading* flag in the *ComponentState* property indicating that the component has been loaded from a stream.

```
procedure Loaded; virtual;
```

Description The *Loaded* method provides an opportunity for a component to initialize itself after all its parts have loaded from a stream.

When a Delphi application loads a form from its form file, for example, it first constructs the form component by calling its *Create* constructor, then reading its property values from the form file, which is a stream. After reading all the property values for all the components, Delphi calls the *Loaded* methods of each component in the order the components were

created. This gives the components a chance to initialize any data that depends on the values of other components or other parts of itself.

Note All references to sibling components are resolved by the time *Loaded* is called. *Loaded* is the first place that sibling pointers can be used after being streamed in.

See also *ComponentState* property, *Create* method, *TReader* object

Notification

Notification forwards notification messages to all owned components.

```
type TOperation = (opInsert, opRemove);
procedure Notification(AComponent: TComponent; Operation: TOperation); virtual;
```

Description The *Notification* method notifies the component that the component specified by AComponent is about to be inserted or removed, as specified by Operation. By default, components pass along the notification to their owned components, if any.

A component can, if needed, act on the notification that a component is being inserted or removed. In particular, if a component has object fields or properties that contain references to other components, it might check the notifications of component removals and invalidate those references as needed.

See also *FreeNotification* method, *InsertComponent* method, *RemoveComponent* method

QueryInterface

QueryInterface returns a reference to a specified interface, if that interface is supported by the object.an object for an interface implemented by the object.

```
function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
```

Description For components that support interfaces, *QueryInterface* calls the implementation of this method for the interface supported by the component. *QueryInterface* returns a reference to the interface, specified by the *IID* parameter, as the **out** parameter. If the component does not support the interface, the out parameter is **nil**.

See also *Invoke* method, *_AddRef* method, *_Release* method

ReadState

ReadState is a method used by the streaming system to read the component's data from a stream.

```
procedure ReadState(Reader: TReader); virtual;
```

Description *ReadState* is a method that communicates with a reader object, passed in *Reader*, to assure that the component is properly read in from a stream.

The *ReadState* method is part of a sequence of calls used by the Delphi streaming system to read the values of all the component's published properties, other stored data from a stream. It is not necessary to call *ReadState* directly.

Although *ReadState* is virtual, it is rarely overridden. Nevertheless, any descendent classes overriding *ReadState* should end with a call to the inherited *ReadState* method of *TComponent*.

ReadState reads the value of the component's published properties, stored data, and owned components from the reader object passed in Reader.

See also *TReader* object, *TReader.ReadComponent* method

RemoveComponent

RemoveComponent removes the component specified in the *AComponent* parameter from the component's *Components* list.

```
procedure RemoveComponent(AComponent: TComponent);
```

Description Components are automatically inserted and removed when visually manipulating them in the Delphi form designer. Use *RemoveComponent* when manually deleting components from an *Owner* component.

See also *Components* property, *Notification* method, *Owner* property

SafeCallException

SafeCallException is an error handling method used to support OLE exceptions.

```
function SafeCallException(ExceptObject: TObject; ExceptAddr: Pointer): Integer; override;
```

Description *SafeCallException* is used for OLE exception handling support. Classes that implement COM and OLE interfaces should override this method to handle other errors that might occur. COM and OLE classes in Delphi override *SafeCallException*. *TComponent* calls the implementation of this method for the interface supported by the component, if one exists. If the component does not support interfaces, this method calls the inherited method, which returns E_UNEXPECTED. This is a default return value that is appropriate for classes that do not support any interfaces.

See also *TObject.SafeCallException* method

SetAncestor

SetAncestor sets or clears the *ComponentState* property's *csAncestor* flag for the component where the method is executed and all components it owns.

```
procedure SetAncestor(Value: Boolean);
```

Description *SetAncestor* is used internally for the Delphi streaming system. It is not necessary to call *SetAncestor* directly.

SetAncestor sets or clears the *csAncestor* flag which is used to determine whether or not the component was introduced in an ancestor form. *SetAncestor* then calls the *SetAncestor* methods of any owned components, passing Value, so that the owned components' *ComponentState* properties will be synchronized with the owner's.

See also *ComponentState* property, *Owner* property

SetChildOrder

SetChildOrder does nothing in *TComponent*.

```
procedure SetChildOrder(Child: TComponent; Order: Integer); dynamic;
```

Description *SetChildOrder* is introduced in *TComponent* for convenience for the Delphi streaming system. Descendent classes override *SetChildOrder* to change the order in which the child appears in the child component list returned by *GetChildren*.

See also *TWinControl.SetChildOrder* method

SetDesigning

SetDesigning ensures that components inserted at design time have their design-mode flag set.

```
procedure SetDesigning(Value: Boolean);
```

Description *SetDesigning* is used internally for the Delphi Form designer. It is not necessary to call *SetDesigning* directly.

SetDesigning sets the *csDesigning* option in the *ComponentState* property if *Value* is *True*; otherwise, it removes *csDesigning*. *SetDesigning* then calls the *SetDesigning* methods of any owned components, passing *Value*, so that the owned components' *ComponentState* properties will be synchronized with the owner's.

The *InsertComponent* and *RemoveComponent* methods call *SetDesigning* for inserted or removed components to ensure that their design-mode flags are set properly.

See also *ComponentState* property

SetName

SetName is the property-access method for setting the value of the *Name* property.

```
procedure SetName(const NewName: TComponentName); virtual;
```

Description *SetName* is the virtual method used as the write part of the *Name* property. It calls the *ChangeName* method, which performs the actual name change. *ChangeName* is not virtual; override *SetName* instead when modifying the behavior of the *Name* property.

See also *ChangeName* method, *Name* property, *ValidateRename* method

SetParentComponent

SetParentComponent does nothing in *TComponent*.

```
procedure SetParentComponent(Value: TComponent); dynamic;
```

Description *SetParentComponent* was introduced in *TComponent* for convenience in the Delphi streaming system. Descendent classes override *SetParentComponent* to change the value that *GetParentComponent* returns to match Value.

See also *GetParent* method, *HasParent* method, *TControl.SetParentComponent* method

Updated

Updated clears the *csUpdating* state in the component's *ComponentState* property to indicate that the component has finished updating.

```
procedure Updated; dynamic;
```

Description *Updated* is used internally in Delphi to modify the *ComponentState* property. It is not necessary to call *Updated* directly. A call to *Updated* always follows a call to *Updating*, which sets the flag.

See also *ComponentState* property, *Updating* method

UpdateRegistry

UpdateRegistry is for internal use only on components that implement COM interfaces.

```
class procedure UpdateRegistry(Register: Boolean; const ClassID, ProgID: string); dynamic;
```

Description *UpdateRegistry* is for internal use only. Do not call *UpdateRegistry* directly.

Updating

Updating sets the *csUpdating* state in the component's *ComponentState* property to indicate that the component is about to be updated.

```
procedure Updating; dynamic;
```

Description *Updating* is used internally in Delphi to modify the *ComponentState* property. It is not necessary to call *Updating* directly. A call to *Updating* should always be followed by a call to *Updated*, which clears the flag.

See also *ComponentState* property, *Updated* method

ValidateContainer

ValidateContainer determines whether an object can be inserted into a container object.

```
procedure ValidateContainer(AComponent: TComponent); dynamic;
```

Description *ValidateContainer* is called by a component when that component is about to be inserted into a container object. By default *ValidateContainer* allows any object to be inserted into a container. Descendent components can override *ValidateContainer* to disallow a component from being inserted.

See also *ValidateInsert* method

ValidateInsert

ValidateInsert does nothing in *TComponent*.

```
procedure ValidateInsert(AComponent: TComponent); dynamic;
```

Description *ValidateInsert* is introduced in *TComponent* so that descendent classes can override it to disallow a component from accepting an object being inserted into it. By default *ValidateInsert* allows any object to be inserted into the component. If a component needs to validate only certain objects, descendent classes can override *ValidateInsert* to filter out those objects.

See also *ValidateContainer* method

ValidateRename

ValidateRename ensures that renaming an owned component does not create name conflict.

```
procedure ValidateRename(AComponent: TComponent; const CurName, NewName: string);
  virtual;
```

Description *ValidateRename* checks to see if a component can rename one of its owned components, passed in *AComponent*, from its current name (*CurName*) to the string passed in *NewName*. If *AComponent* is **nil** or *NewName* is already the name of a component in the Components list, *ValidateRename* raises an *EComponentError* exception.

ValidateRename is used internally by Delphi when the *Name* property is modified. It is not necessary to call it directly.

See also *ChangeName* method, *Name* property, *SetName* method

WriteState

WriteState is a method used by the streaming system to write the component's data to a stream.

```
procedure WriteState(Writer: TWriter); virtual;
```

Description *WriteState* is an interactive method that communicates with a writer object, passed in Writer, to assure that the component is properly written out.

The *WriteState* method is part of a sequence of calls used by the Delphi streaming system to write the values of all the component's published properties, other stored data to a stream. It is not necessary to call *WriteState* directly.

Although *WriteState* is virtual, it is rarely overridden. Nevertheless, any descendent classes overriding *WriteState* should end with a call to the inherited *WriteState* method of *TComponent*.

See also *Writer* object, *TWriter.WriteComponent* method

_AddRef

_AddRef increments the reference count for the number of interface references dependent upon the component.

```
function _AddRef: Integer; stdcall;
```

Description *_AddRef* is the implementation of the *IUnknown* interface method, *AddRef*, defined by COM. For components that support interfaces, *_AddRef* calls the implementation of this method for the interface supported by the component. *_AddRef* returns the resulting value of the reference count for the component.

See also *Invoke* method, *QueryInterface* method, *_Release* method

_Release

_Release decrements the reference count for the calling interface on a object.

```
function _Release: Integer; stdcall;
```

Description *_Release* is the implementation for the *IUnknown* interface method, *Release*, defined by COM. For components that support interfaces, *_Release* calls the implementation of this method for the interface supported by the component. If the reference count on the object falls to 0, the object is freed from memory.

_Release returns the resulting value of the reference count for the component.

See also *Invoke* method, *QueryInterface* method, *_AddRef* method

TControl

Unit: Controls

TObject • TPersistent • TComponent • TControl

TControl is the abstract base class for all visual components.

Description Controls are visual components, meaning the user can see them and manipulate them at runtime. All controls have properties, methods, and events in common that are specific to the visual aspect of controls, such as the position of the control, the cursor or hint associated with the control's window, methods to paint or move the control, and events to respond to mouse actions.

TControl has many protected properties and methods that are used or published by its descendants.

See also *TWinControl* object

Properties

The following table lists the properties of *TControl*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Caption	protected		TControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TControl
ClientRect	public, read-only		TControl
ClientWidth	public		TControl
Color	protected	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
DesktopFont	protected	False	TControl
DragCursor	protected	crDrag	TControl
DragMode	protected	dmManual	TControl
Enabled	public	True	TControl
Font	protected		TControl
Height	published		TControl
Hint	published		TControl
IsControl	protected		TControl
Left	published		TControl
MouseCapture	protected		TControl

Property	Visibility	Default	Documented in
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	True	TControl
ParentFont	protected	True	TControl
ParentShowHint	protected	True	TControl
PopupMenu	protected		TControl
ScalingFlags	protected		TControl
ShowHint	public		TControl
Tag	published	0	TComponent
Text	protected		TControl
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl
WindowText	protected		TControl

Align

Align determines how the control aligns within its container (or parent control).

```
TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient);
```

```
property Align: TAlign;
```

Description

Use *Align* to align a control to the top, bottom, left, or right of a form or panel and have it remain there even if the size of the form, panel, or component that contains the control changes.

For example, to use a panel component with various controls on it as a tool palette, change the panel's *Align* value to *alLeft*. The value of *alLeft* for the *Align* property of the panel guarantees that the tool palette remains on the left side of the form and always equals the client height of the form.

The default value of *Align* is *alNone*, which means a control remains where it is positioned on a form or panel. Specify another value to align the control. Use one of these possible values:

Value	Meaning
alNone	The component remains where it was placed on the form or panel. This is the default value.
alTop	The component moves to the top of the form and resizes to fill the width of the form. The height of the component is not affected.
alBottom	The component moves to the bottom of the form and resizes to fill the width of the form. The height of the component is not affected.

Value	Meaning
alLeft	The component moves to the left side of the form and resizes to fill the height of the form. The width of the component is not affected.
alRight	The component moves to the right side of the form and resizes to fill the height of the form. The width of the component is not affected.
alClient	The component resizes to fill the client area of a form. If a component already occupies part of the client area, the component resizes to fit within the remaining client area.

Note If *Align* is set to *alClient*, the control fills the entire client area so that it is impossible to select the parent form by clicking on it. In this case, select the parent by selecting the control on the form and pressing *Esc*, or by using the Object Inspector.

Any number of child components within a single parent can have the same *Align* value, in which case they stack up along the edge of the parent form or panel. The child controls stack up in z-order. To adjust the order in which the controls stack up, drag the controls into their desired positions.

See also *Parent* property

BoundsRect

BoundsRect is the bounding rectangle of the control, expressed in the coordinate system of the parent control.

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;

property BoundsRect: TRect;
```

Description Use *BoundsRect* as a quick way to obtain the pixel locations of all corners of the control all at once.

For example, the statement

```
R := Control.BoundsRect;
```

corresponds to

```
R.Left := Control.Left;
R.Top := Control.Top;
R.Right := Control.Left + Control.Width;
R.Bottom := Control.Top + Control.Height;
```

The *TRect* type defines a rectangle. The coordinates are specified both as four separate integers representing the pixel locations of the left, top, right, and bottom sides, and as two points representing the pixel locations of the top left and bottom right corners. The origin of the pixel coordinate system is in the top left corner of the parent window.

See also *Height* property, *Left* property, *Top* property, *Width* property

Caption

Caption specifies a text string that identifies the control to the user.

```
type TCaption = type string;
```

```
property Caption: TCaption;
```

Description Use *Caption* to associate the control with a text string that identifies the control.

For most controls, the *Caption* is the window title and it appears as a label for the control.

To underline a character in a *Caption* that labels a component, include an ampersand (&) before the character. This type of character is called an accelerator character. The user can then select the component by pressing Alt while typing the underlined character. To display an ampersand character in the caption, use two ampersands (&&).

See also *Name* property, *Text* property

ClientHeight

ClientHeight is the height of the control's client area in pixels.

```
property ClientHeight: Integer;
```

Description Use *ClientHeight* to read or change the height of the control's client area.

For TControl, ClientHeight is the same as *Height*. Derived classes may implement a *ClientHeight* property that differs from *Height*. For example, the *ClientHeight* of a form is the value of the *Height* property minus the height of the title bar, resize border, and scrollbars.

See also *ClientOrigin* property, *ClientRect* property, *ClientWidth* property, *Height* property

ClientOrigin

ClientOrigin is the screen coordinates (in pixels) of the top left corner of a control's client area.

```
TPoint = record
  X: Longint;
  Y: Longint;
end;
```

```
property ClientOrigin: TPoint;
```

Description Read *ClientOrigin* to locate the top-left corner of the control's client area. *ClientOrigin* returns *X* and *Y* coordinates in a record of type *TPoint*, with the

origin in the top left corner. *X* specifies the horizontal coordinate of the point, *Y* specifies the vertical coordinate.

The screen coordinates of a control that is descended from *TControl* and not *TWinControl* are actually the screen coordinates of the control's parent added to its *Left* and *Top* properties. If the control doesn't have a parent, an *EInvalidOperation* exception is raised when *ClientOrigin* is read.

See also *ClientHeight* property, *ClientRect* property, *ClientWidth* property, *EInvalidOperation* object

ClientRect

ClientRect returns the size (in pixels) of a control client area.

```
TPoint = record
  X: Longint;
  Y: Longint;
end;

TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;

property ClientRect: TRect;
```

Description Read *ClientRect* to find the size of the client area of a control. *ClientRect* returns a rectangle with its *Top* and *Left* fields set to zero, and its *Bottom* and *Right* fields set to the control's *Height* and *Width*, respectively. *ClientRect* is equivalent to *Rect*(0, 0, *ClientWidth*, *ClientHeight*).

The *TRect* type defines a rectangle. The coordinates are specified both as four separate integers representing the pixel locations of the left, top, right, and bottom sides, and as two points representing the pixel locations of the top left and bottom right corners.

See also *BoundsRect* property, *ClientHeight* property, *ClientWidth* property, *Height* property, *Left* property, *Top* property, *Width* property

ClientWidth

ClientWidth is the horizontal size of the control's client area in pixels.

```
property ClientWidth: Integer;
```

Description Use *ClientWidth* to read or change the width of the control's client area. *ClientWidth* is equivalent to *ClientRect.Right*.

See also *ClientHeight* property, *ClientOrigin* property, *ClientRect* property, *Width* property

Color

Color is the background color of the control.

property Color: TColor;

Description Use *Color* to read or change the background color of the control. *Color* can be an RGB value or one of the constants defined in the Graphics unit. If *Color* is an RGB value, the low three bytes represent RGB color intensities for blue, green, and red, respectively.

These are the predefined constants that can be used for *Color*:

Value	Meaning
clBlack	Black
clMaroon	Maroon
clGreen	Green
clOlive	Olive green
clNavy	Navy blue
clPurple	Purple
clTeal	Teal
clGray	Gray
clSilver	Silver
clRed	Red
clLime	Lime green
clBlue	Blue
clFuchsia	Fuchsia
clAqua	Aqua
clWhite	White
clBackground	Current background color of the Windows desktop
clActiveCaption	Current color of the title bar of the active window
clInactiveCaption	Current color of the title bar of inactive windows
clMenu	Current background color of menus
clWindow	Current background color of windows
clWindowFrame	Current color of window frames
clMenuText	Current color of text on menus
clWindowText	Current color of text in windows
clCaptionText	Current color of the text on the title bar of the active window
clActiveBorder	Current border color of the active window
clInactiveBorder	Current border color of inactive windows
clAppWorkSpace	Current color of the application workspace
clHighlight	Current background color of selected text
clHightlightText	Current color of selected text
clBtnFace	Current color of a button face

Value	Meaning
clBtnShadow	Current color of a shadow cast by a button
clGrayText	Current color of text that is dimmed
clBtnText	Current color of text on a button
clInactiveCaptionText	Current color of the text on the title bar of an inactive window
clBtnHighlight	Current color of the highlighting on a button
cl3DDkShadow	Windows 95 or NT 4.0 only: Dark shadow for three-dimensional display elements
cl3DLight	Windows 95 or NT 4.0 only: Light color for three-dimensional display elements (for edges facing the light source)
clInfoText	Windows 95 or NT 4.0 only: Text color for tool tip controls
clInfoBk	Windows 95 or NT 4.0 only: Background color for tool tip controls

The second half of the colors listed here are Windows system colors. The color that appears depends on the color scheme users are using for Windows. Users can change these colors using the Control Panel. The actual color that appears will vary from system to system. For example, the *clBtnFace* system color might be gray in one user's color scheme, and tan in another user's color scheme.

Note If a control's *ParentColor* property is *True*, then changing the *Color* property of the control's parent automatically changes the *Color* property of the control. When you assign a value to a control's *Color* property, the control's *ParentColor* property is automatically set to *False*.

See also *ParentColor* property

ControlState

ControlState holds a set of values indicating the current state of a control at runtime.

```
type
  TControlState = set of (csLButtonDown, csClicked, csPalette, csReadingState,
    csAlignmentNeeded, csFocusing, csCreating);

property ControlState: TControlState;
```

Description Read *ControlState* to find out various conditions that affect the control such as whether it has been clicked or needs alignment. *ControlState* reflects transient conditions of an instance of the control, as opposed to attributes of the

control class as a whole. *ControlState* consists of a set of flags drawn from the following values:

Flag	Meaning
csLButtonDown	The left mouse button was clicked and not yet released. This is set for all mouse-down events.
csClicked	The same as csLButtonDown, but only set if ControlStyle contains *csClickEvents*, meaning that mouse-down events are interpreted as clicks.
csPalette	A WM_PALETTCHANGED message was received by the control or one of its parents.
csReadingState	The control is reading its state from a stream.
csAlignmentNeeded	The control needs to realign itself when alignment is re-enabled.
csFocusing	The application is processing messages intended to give the control focus. This does not guarantee the control will receive focus, but prevents recursive calls.
csCreating	The control and/or its owner and subcontrols is being created. This flag clears when all have finished creating.
csPaintCopy	The control is being replicated, meaning a copy of the control is being painted. The ControlStyle flag csReplicatable must be set for this state to occur.

The flags in *ControlState* are specific to controls, and are in addition to the state flags in the *ComponentState* property, which apply to all components.

ControlState is primarily used by component writers in the implementation of components derived from *TControl*.

See also *ControlStyle* property, *TComponent.ComponentState* property

C

ControlStyle

ControlStyle holds a set of values that determine style characteristics of the control.

```
type
  TControlStyle = set of (csAcceptsControls, csCaptureMouse, csDesignInteractive,
    csClickEvents, csFramed, csSetCaption, csOpaque, csDoubleClicks, csFixedWidth,
    csFixedHeight, csNoDesignVisible, csReplicatable, csNoStdEvents, csDisplayDragImage);

property ControlStyle: TControlStyle;
```

Description Use *ControlStyle* to determine various attributes of the control, such as whether the control can capture the mouse or has a fixed size. The

ControlStyle property contains a set of style flags indicating these attributes. The following table lists the flags and their meanings:

Flag	Meaning
csAcceptsControls	The control becomes the parent of any controls dropped on it at design time.
csCaptureMouse	The control captures mouse events when it is clicked.
csDesignInteractive	The control maps right mouse-button clicks at design time into left-button clicks to manipulate the control.
csClickEvents	The control can receive and respond to mouse clicks.
csFramed	The control has a 3D frame.
csSetCaption	The control should change its caption to match the Name property if the caption has not been explicitly set to something else.
csOpaque	The control completely fills its client rectangle.
csDoubleClicks	The control can receive and respond to double-click messages. Otherwise, map double-clicks into clicks.
csFixedWidth	The width of the control does not vary or scale.
csFixedHeight	The height of the control does not vary or scale.
csNoDesignVisible	The control is not visible at design time.
csReplicatable	The control can be copied using the *PaintTo* method to draw its image to an arbitrary canvas.
csNoStdEvents	Standard events such as mouse, key, and click events are ignored. Use this setting if your code doesn't need to respond to these events and your application will run faster.
csDisplayDragImage	The control can display an image from an image list when the mouse is dragged over the control. Use this setting if the control implements an image list for when the mouse is dragged over it.

ControlStyle generally describes an entire class, which all instances reflect. The flags do not change among different instances of a type of control, nor do they vary at runtime. Such transient conditions are indicated by the *ControlState* property.

The *Create* method of *TControl* initializes *ControlStyle* to [*csCaptureMouse, csClickEvents, csSetCaption, csDoubleClicks*].

ControlStyle is primarily used by component writers in the implementation of components derived from *TControl*.

See also *ControlState* property, *Create* method

Cursor

Cursor specifies the image used to represent the mouse pointer when it passes into the region covered by the control.

```
type TCursor = -32768..32767;
```

```
property Cursor: TCursor;
```

Description Change the value of *Cursor* to provide feedback to the user about what action is appropriate when the mouse pointer enters the control.

The value of *Cursor* is the index of the cursor in the list of cursors maintained by the global variable, *Screen*. In addition to the built-in cursors provided by *TScreen*, applications can add custom cursors to the list. These are the built-in values provided by *TScreen*:

Value	Image	Value	Image
crDefault	Whatever cursor is the default for the window class. Usually *crArrow*.	*crHourGlass*	
crNone		*crDrag*	
crArrow		*crNoDrop*	
crCross		*crHSplit*	
crIBeam		*crVSplit*	
crSize		*crMultiDrag*	
crSizeNESW		*crSQLWait*	
crSizeNS		*crNo*	
crSizeNWSE		*crAppStart*	
crSizeWE		*crHelp*	
crUpArrow		*crHandPoint*	

Note: To learn how to make a custom cursor available for the *Cursor* property, see the *Cursors* property of the *TScreen* object.

See also *DragCursor* property, *TScreen.Cursors* property

DesktopFont

DesktopFont specifies whether the control uses the Windows icon font when writing text.

```
property DesktopFont: Boolean;
```

Description Set *DesktopFont* to *True* to specify that the control should use the Windows icon font for its *Font* property. The Windows icon font is the value of the *IconFont* property of the global Screen variable.

When *DesktopFont* is *True*, the *Font* property of the control updates every time the Windows icon font is changed. This font can be changed by setting the *IconFont* property of the global *Screen* variable, or by the actions of other programs. Setting the *Font* property to *Screen.IconFont* will not allow controls to respond to changes in the Windows icon font in this manner.

See also *Font* property, *TScreen.IconFont* property

DragCursor

DragCursor contains the image used to represent the mouse pointer when the control is being dragged.

```
type TCursor = -32768..32767;

property Cursor: TCursor;
```

Description Use the *DragCursor* property to change the cursor image presented when the control is being dragged. The possible values for the *DragCursor* property are the same as those for the *Cursor* property. See the table in the *Cursor* property entry.

Note: To learn how to make a custom cursor available for the *DragCursor* property, see the *Cursors* property of the *TScreen* component.

See also *Cursor* property, *TScreen.Cursors* property

DragMode

DragMode determines the drag and drop behavior of a control.

```
type TDragMode = (dmManual, dmAutomatic);

property DragMode: TDragMode;
```

Description Use the *DragMode* property to control when the user is able to drag the control. Disable the drag and drop capability at runtime by changing the *DragMode* property value to *dmManual*.

These are the possible values:

Value	Meaning
dmAutomatic	If *dmAutomatic* is selected, the control is ready to be dragged; the user just clicks and drags it.
dmManual	If *dmManual* is selected, the control can't be dragged until the application calls the *BeginDrag* method.

See also *BeginDrag* method, *DragCursor* property, *EndDrag* method, *OnDragDrop* event, *OnDragOver* event, *OnEndDrag* event, *OnStartDrag* event

Enabled

Enabled controls whether the control responds to mouse, keyboard, and timer events.

property Enabled: Boolean;

Description Use *Enabled* to change the availability of the control to the user. To disable a control, set *Enabled* to *False*. Disabled controls appear dimmed. If *Enabled* is *False*, the control ignores mouse and keyboard events, and in the case of a timer control, the *OnTimer* event.

To re-enable a control, set *Enabled* to *True*. The control is no longer dimmed, and the user can use the control.

See also *OnTimer* event, *Visible* property

Font

Font controls the attributes of text written on or in the control.

property Font: TFont;

Description To change to a new font, specify a new *TFont* object. To modify a font, change the value of the *Color*, *Height*, *Name*, *Pitch*, *Size*, or *Style* of the *TFont* object.

See also *TFont* object

Height

Height is the vertical size of the control in pixels.

property Height: Integer;

Description Use the *Height* property to read or change the height of the control.

See also *ClientHeight* property, *Width* property

Hint

Hint contains the text string that can appear when the user moves the mouse pointer over the control.

property Hint: **string**;

Description

Use the *Hint* property to provide a string of help text either as a Help Hint, or as help text on a particular location such as a status bar.

A Help Hint is a box containing help text that appears for a control when the user moves the mouse pointer over the control and pauses momentarily. To set up Help Hints:

1. Specify the *Hint* property for each control that a Help Hint should appear for.
2. Set the *ShowHint* property of each appropriate control to *True*, or set the *ParentShowHint* property of all controls to *True* and set the *ShowHint* property of the form to *True*.
3. At runtime, set the value of the application's *ShowHint* property to *True*.

To show the *Hint* on a status bar or other location, use the *OnHint* event handler of the application. The application's *OnHint* event occurs when the mouse pointer moves over the control. Write an *OnHint* event handler to display the hint as the caption of a panel component that is used as a status bar.

Specify a hint to be used for both a Help Hint box and by an *OnHint* handler by specifying two values separated by a | character (the vertical bar "or" symbol). For example,

```
Edit1.Hint := 'Name|Enter Name in the edit box';
```

The 'Name' string appears in the Help Hint box, and the 'Enter full name' string appears as specified in the *OnHint* event handler.

If you specify just one value, it can be used as both a Help Hint and as the *Hint* property of the application. If the application's *ShowHint* property is *False*, the Help Hint won't appear, but the *OnHint* event handler will still be called.

If a control has no *Hint* value specified, but its parent control does, the control uses the *Hint* value of the parent control as long as the control's *ShowHint* property is *True*.

See also

GetLongHint function, *GetShortHint* function, *ParentShowHint* property, *ShowHint* property, *TApplication.Hint* property, *TApplication.OnHint* event, *TApplication.ShowHint* property

IsControl

IsControl determines whether a form stores its form-specific properties to a stream.

property IsControl: Boolean;

Description Use *IsControl* to use a form as a complex control, rather than as a desktop form. *IsControl* affects the way the form gets saved. When *IsControl* is *True*, form-specific properties do not get saved with the form. When *IsControl* is *False*, the form-specific properties get saved along with the other properties.

IsControl permits you to use the forms designer to create complex controls such as panels, by creating those controls as forms, placing and naming their contained controls, and attaching code to events. After saving the form, edit the form as text, and in the text version set *IsControl* to *True*. The next time the form is reloaded and saved, only those form properties appropriate to use as a control are stored, not the properties specific to *TForm*. Next, edit the form file as text, changing the type of the component from *TForm* to the desired control type, such as *TPanel*.

See also *TForm* object

Left

Left is the horizontal coordinate of the left edge of a component relative to the form in pixels.

property Left: Integer;

Description Use the *Left* property to determine where the left side of the control begins or to reposition the left side of the control.

If the control is contained in another control, such as a *TPanel*, the *Left* and *Top* properties are relative to the parent control. If the control is contained directly by the form, the properties values are relative to the form. For forms, the value of the *Left* property is relative to the screen in pixels.

See also *Top* property

C

MouseCapture

MouseCapture indicates whether the control has "captured" mouse events.

property MouseCapture: Boolean;

Description Use *MouseCapture* to determine whether a control has captured the mouse. The capture control is the control to which all subsequent mouse events go until the user releases the mouse button.

A control becomes the capture control when the user drags an item from it. In addition, if the control has the *csCaptureMouse* flag set in its *ControlStyle* property, it becomes the capture control when the user presses the left mouse button over it, until the user releases the mouse button.

See also *ControlStyle* property, *GetCaptureControl* function, *SetCaptureControl* procedure

Name

Name is the name of the control as referenced in the application's code.

```
type TComponentName: string;
```

```
property Name: TComponentName;
```

Description Use the *Name* property to assign a new name to the control or to find out what the name of the control is.

By default, the IDE assigns sequential names based on the type of the control, such as 'Button1', 'Button2', and so on. Change these to more meaningful names that make the application's code more readable. The *Name* must be a valid Object Pascal identifier. If the control's *Caption* has not already been modified, the *Caption* is changed to match the new control name.

Note: Change control names only at design time.

See also *Caption* property

Parent

Parent refers to the parent of the control and determines where the control is displayed.

```
property Parent: TWinControl;
```

Description Use the *Parent* property to determine the parent of this control or to assign a control to be the parent of this control. When creating a new control, always assign a *Parent* property value for the new control. Usually, this is a form, panel, group box, or some control that is designed to contain another. Changing the parent of a control moves the control onscreen so that it is displayed within the parent.

The parent of a control is the windowed control that contains the control. If one control (parent) contains others, the contained controls are child controls of the parent. For example, if an application includes three radio buttons in a group box, the group box is the parent of the three radio buttons, and the radio buttons are the child controls of the group box.

Don't confuse the *Parent* property declared in *TControl* with the *Owner* property declared in *TComponent*. The *Parent* of a control is always a windowed control that visually contains the control. The *Owner* of a component is the component that was passed as a parameter in the constructor and that uses the component as a constituent part.

See also *TComponent.Owner* property, *TWinControl.Controls* property

ParentColor

ParentColor determines where a control looks for its color information.

```
property ParentColor: Boolean;
```

Description To have a control use the same color as its parent control, set *ParentColor* to *True*. For example, if the *ParentColor* of a list box on a panel is *True*, the list box has the same color as the panel. If *ParentColor* is *False*, the control uses its own *Color* property.

Set *ParentColor* to *True* for all controls in order to ensure that all the controls on a form have a uniform appearance. For example, if *ParentColor* is *True* for all controls in a form, changing the background color of the form to gray causes all the controls on the form to also have a gray background.

To specify a different color for a particular control, assign the desired color to that control's *Color* property, and *ParentColor* becomes *False* automatically.

See also *Color* property, *Parent* property, *ParentFont* property, *ParentShowHint* property

ParentFont

ParentFont determines where a control looks for its font information.

```
property ParentFont: Boolean;
```

Description To have a control use the same font as its parent control, set *ParentFont* to *True*. For example, if the *ParentFont* of a list box on a panel is *True*, the list box uses the same font as the panel. If *ParentFont* is *False*, the control uses its own *Font* property.

Set *ParentFont to True for all controls in order to* ensure that all the controls on a form have a uniform appearance. For example, if *ParentFont* is *True* for all controls in a form, changing the form's *Font* property to 12-point Courier causes all controls on the form to use that font.

To specify a different font for a particular control, change the control's *Font* property, and *ParentFont* becomes *False* automatically.

When the *ParentFont* is *True* for a form, the form uses the value of the application's *Font* property.

See also *Font* property, *Parent* property, *ParentColor* property, *ParentShowHint* property

ParentShowHint

ParentShowHint determines where a control looks to find out if the Help Hint, specified as the value of the *Hint* property for the control, should be shown.

```
property ParentShowHint: Boolean;
```

Description Use *ParentShowHint* to ensure that all the controls on a form either uniformly show their Help Hints or uniformly don't show them.

If *ParentShowHint* is *True*, the control uses the *ShowHint* property value of its parent. If *ParentShowHint* is *False*, the control uses the value of its own *ShowHint* property.

To provide Help Hints for only selected controls on a form, set the *ShowHint* property for those controls that should have Help Hints to *True*, and *ParentShowHint* becomes *False* automatically.

Note Enable or disable all Help Hints for the entire application using the *ShowHint* property of the application object.

See also *Hint* property, *Parent* property, *ParentColor* property, *ParentFont* property, *ShowHint* property

PopupMenu

PopupMenu identifies the pop-up menu associated with the control.

```
property PopupMenu: TPopupMenu;
```

Description Assign a value to *PopupMenu to make* a pop-up menu appear when the user selects the control and clicks the right mouse button. If the *TPopupMenu*'s *AutoPopup* property is *True*, the pop-up menu appears automatically. If the menu's *AutoPopup* property is *False*, display the menu with a call to its *Popup* method.

See also *TPopupMenu* object, *TPopupMenu.AutoPopup* property, *TPopupMenu.Popup* method

ScalingFlags

ScalingFlags reflects which attributes of the control have not yet been scaled.

```
type TScalingFlags = set of (sfLeft, sfTop, sfWidth, sfHeight, sfFont);
property ScalingFlags: TScalingFlags;
```

Description *ScalingFlags* is an internal set of flags used primarily in scaling inherited forms and their controls. Applications should not use this property without a serious need.

These are the flags and their meanings:

Flag	Meaning
sfLeft	Left side of control not yet scaled.
sfTop	Top of control not yet scaled.
sfWidth	Width of control not yet scaled.
sfHeight	Height of control not yet scaled.
sfFont	Font of control not yet scaled.

See also *ChangeScale* method

ShowHint

ShowHint determines if the control should display a Help Hint when the mouse pointer rests momentarily on the control.

```
property ShowHint: Boolean;
```

Description The Help Hint is the value of the *Hint* property, which is displayed in a box just beneath the control. Use *ShowHint* to determine whether a Help Hint appears for the control.

To enable Help Hint for a particular control, the application *ShowHint* must be *True* and either

- the control's own *ShowHint* property must be *True, or*
- the control's *ParentShowHint* property must be *True and* its parent's *ShowHint* property must be *True.*

For example, imagine a check box within a group box. If the *ShowHint* property of the group box is *True* and the *ParentShowHint* property of the check box is *True,* but the *ShowHint* property of the check box is *False,* the check box will still display its Help Hint.

Changing the *ShowHint* value automatically sets the *ParentShowHint* property to *False.*

See also *Hint* property, *ParentShowHint* property, *TApplication.ShowHint* property

Text

Text contains a text string associated with the control.

```
type TCaption = type string;
property Text: TCaption;
```

Description Use the *Text* property to read the *Text* of the control or specify a new string for the *Text* value. By default, *Text* is the control name. For edit controls and memos, the *Text* appears within the control. For combo boxes, the *Text* is the content of the edit control portion of the combo box.

See also *Caption* property

Top

Top is the Y coordinate of the top left corner of a control, relative to its parent or containing control in pixels.

```
property Top: Integer;
```

Description Use the *Left* property to locate the top of the control or reposition the control to a different Y coordinate. The *Top* property, like the *Left* property, is the position of the control relative to its container. Thus, if a control is contained in a *TPanel,* the *Left* and *Top* properties are relative to the panel. If the control

is contained directly by the form, it is relative to the form. For forms, the value of the *Left* property is relative to the screen in pixels.

See also *Left* property

Visible

Visible determines whether the component appears onscreen.

```
property Visible: Boolean;
```

Description Use the *Visible* property to control the visibility of the control at runtime. If *Visible* is *True*, the control appears. If *Visible* is *False*, the control is not visible.

Calling the *Show* method sets the control's *Visible* property to *True*. Calling the *Hide* method sets it to *False*.

See also *Enabled* property, *Hide* method, *Show* method

Width

Width determines the horizontal size of the control or form in pixels.

```
property Width: Integer;
```

Description Use the *Width* property to read or change the width of the control.

Note: For tab sheet controls, changing this property at runtime has no effect.

See also *ClientWidth* property, *Height* property, *Left* property

WindowProc

WindowProc contains the window procedure that responds to messages sent to the control.

```
type TWndMethod = procedure(var Message: TMessage) of object;
```

```
property WindowProc: TWndMethod;
```

Description Use the *WindowProc* property to temporarily replace or subclass the window procedure of the control. Component writers that are customizing the window procedure for a descendent class should override the *WndProc* method instead.

Before assigning a new value to *WindowProc*, store the original value. *WindowProc* is initially set to the *WndProc* method, so if the value has not been changed since then, the original value need not be stored. Within the procedure that is used as the new value for *WindowProc*, pass any unhandled messages to the original procedure that was the value of *WindowProc*. After completing any specialized message handling, restore the value of *WindowProc* to the original procedure.

See also *WndProc* method

WindowText

WindowText contains the text string of the control.

```
property WindowText: PChar;
```

Description Use *WindowText* to determine the text associated with the control. When setting *WindowText*, the control frees the memory associated with the previous value of *WindowText* before assigning a new value. Any memory associated with *WindowText* is freed when the control is destroyed.

By default, *WindowText* is the same string as the *Text* property. Descendants of *TControl* can override the *Text* property to implement a different value.

For an edit box, the *WindowText* is the content of the control. For a combo box, the text string is the content of the edit box part of the combo box. For a button, the string is the button name. For all other controls, the string is the window title.

See also *Caption* property, *Text* property

Events

The following table lists the events of *TControl*:

Event	Visibility	Documented in
OnClick	protected	TControl
OnDblClick	protected	TControl
OnDragDrop	protected	TControl
OnDragOver	protected	TControl
OnEndDrag	protected	TControl
OnMouseDown	protected	TControl
OnMouseMove	protected	TControl
OnMouseUp	protected	TControl
OnStartDrag	protected	TControl

C

OnClick

OnClick occurs when the user clicks the control.

```
type TNotifyEvent = procedure (Sender: TObject) of object;
property OnClick: TNotifyEvent;
```

Description Use the *OnClick* event handler to write code that responds to the click event.

Usually *OnClick* occurs because the user presses and releases the primary mouse button with the mouse pointer over the control. This event can also occur when

- The user selects an item in a grid, outline, list, or combo box by pressing an arrow key.
- The user presses *Spacebar* while a button or check box has focus.
- The user presses *Enter* when the active form has a default button (specified by the *Default* property).
- The user presses *Esc* when the active form has a cancel button (specified by the *Cancel* property).
- The user presses the accelerator key for a button or check box. For example, if the value of the *Caption* property of a check box is '&Bold', the B is underlined at runtime and the *OnClick* event of the check box is triggered when the user presses *Alt+B*.
- The *Checked* property of a radio button is set to *True*.
- The value of the *Checked* property of a check box is changed.
- The *Click* method of a menu item is called.

For a form, an *OnClick* event occurs when the user clicks a blank area of the form or on a disabled component.

See also *OnDblClick* event, *TButton.Cancel* property, *TButton.Default* property, *TMenuItem.Click* method

OnDblClick

OnDblClick occurs when the user double-clicks the primary mouse button when the mouse pointer is over the control.

```
type TNotifyEvent = procedure (Sender: TObject) of object;

property OnDblClick: TNotifyEvent;
```

Description Use the *OnDblClick* event to write code that responds to the double-click event.

See also *OnClick* event

OnDragDrop

OnDragDrop occurs when the user drops an object being dragged.

```
type TDragDropEvent = procedure(Sender, Source: TObject; X, Y: Integer) of object;

property OnDragDrop: TDragDropEvent;
```

Description Use the *OnDragDrop* event handler to specify what you want to happen when the user drops an object. The *Source* parameter of the *OnDragDrop* event is the object being dropped, and the *Sender* is the control the object is being dropped on. The *X* and *Y* parameters are the coordinates of the mouse positioned over the control.

See also *BeginDrag* method, *DragDrop* method, *Dragging* method, *EndDrag* method, *OnDragOver* event, *OnEndDrag* event, *OnStartDrag* event

OnDragOver

OnDragOver occurs when the user drags an object over a control.

```
type
  TDragState = (dsDragEnter, dsDragLeave, dsDragMove);
  TDragOverEvent = procedure(Sender, Source: TObject; X, Y: Integer; State: TDragState;
    var Accept: Boolean) of object;

property OnDragOver: TDragOverEvent;
```

Description Use an *OnDragOver* event to signal that the control can accept the dragged object so the user can drop it.

Within the *OnDragOver* event handler, change the *Accept* parameter to *False* to reject the dragged object. Leave *Accept* as *True* to allow the user to drop the dragged object on the control.

To change the shape of the cursor, indicating that the control can accept the dragged object if the user drops it, change the value of the *DragCursor* property for the control before the *OnDragOver* event occurs.

The *Source* is the object being dragged, the *Sender* is **self**, the potential drop site, and *X* and *Y* are screen coordinates in pixels. The *TDragState* type specifies the drag state of a dragged control in relationship to another control. These are the possible states:

Value	Meaning
dsDragEnter	The mouse is entering the control.
dsDragMove	The mouse moved while over the control.
dsDragLeave	The mouse is leaving the control.

Note Within the *OnDragOver* event handler, the *Accept* parameter defaults to *True*. However, if an *OnDragOver* event handler is not supplied, the control rejects the dragged object, as if the *Accept* parameter was changed to *False*.

See also *DragCursor* property, *OnDragDrop* event, *OnEndDrag* event, *OnStartDrag* event

OnEndDrag

OnEndDrag occurs when the dragging of an object ends, either by dropping the object or by canceling the dragging.

```
type TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer) of object;

property OnEndDrag: TEndDragEvent;
```

Description Use the *OnEndDrag* event handler to specify any special processing that occurs when dragging stops. The *Sender* is the object being dragged, *Target* is the object *Sender* is dragged to, and *X* and *Y* are screen coordinates in pixels.

The *OnEndDrag* event is received by the *Sender*. If the dragged object was dropped and accepted by a control, the *Target* parameter of the *OnEndDrag* event is set to the object that accepted the drop. If the object was not dropped successfully, the value of *Target* is **nil**.

See also *OnDragDrop* event, *OnDragOver* event, *OnStartDrag* event

OnMouseDown

OnMouseDown occurs when the user presses a mouse button with the mouse pointer over a control.

```
type
  TMouseButton = (mbLeft, mbRight, mbMiddle);
  TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);
  TMouseEvent = procedure (Sender: TObject; Button: TMouseButton; Shift: TShiftState;
    X, Y: Integer) of object;

property OnMouseDown: TMouseEvent;
```

Description Use the *OnMouseDown* event handler to implement any special processing that should occur as a result of pressing a mouse button.

The *OnMouseDown* event handler can respond to left, right, or center mouse button presses and shift key plus mouse-button combinations. Shift keys are the *Shift, Ctrl,* and *Alt* keys. *X* and *Y* are the pixel coordinates of the mouse pointer in the client area of the Sender.

See also *OnClick* event, *OnMouseMove* event, *OnMouseUp* event

OnMouseMove

OnMouseMove occurs when the user moves the mouse pointer while the mouse pointer is over a control.

```
type
  TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);
  TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState; X, Y: Integer) of object;

property OnMouseMove: TMouseMoveEvent;
```

Description Use the *OnMouseMove* event handler to make something happen when the mouse pointer moves within the control.

Use the *Shift* parameter of the *OnMouseDown* event handler, to respond to the state of the shift keys and mouse buttons. Shift keys are the *Shift, Ctrl,* and *Alt* keys or shift key-mouse button combinations. *X* and *Y* are pixel coordinates of the new location of the mouse pointer in the client area of the *Sender*.

See also *OnMouseDown* event, *OnMouseUp* event

OnMouseUp

OnMouseUp occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

```
type
  TMouseButton = (mbLeft, mbRight, mbMiddle);
  TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);
  TMouseEvent = procedure (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y:
    Integer) of object;

property OnMouseUp: TMouseEvent;
```

Description Use the *OnMouseUp* event handler to implement special processing when the user releases a mouse button.

The *OnMouseUp* event handler can respond to left, right, or center mouse button presses and shift key plus mouse-button combinations. Shift keys are the *Shift, Ctrl,* and *Alt* keys. *X* and *Y* are the pixel coordinates of the mouse pointer in the client area of the *Sender*.

See also *OnMouseDown* event, *OnMouseMove* event

OnStartDrag

OnStartDrag occurs when the user has begun to drag the control or an object it contains by left-clicking on the control and holding the mouse button down.

```
type TStartDragEvent = procedure (Sender: TObject; var DragObject: TDragObject) of object;

property OnStartDrag: TStartDragEvent;
```

Description Use the *OnStartDrag* event handler to implement special processing when the user starts to drag the control or an object it contains. *Sender* is **self**, the control that is about to be dragged, or that contains the object about to be dragged.

The *OnStartDrag* event handler can create a *TDragObject* object for the *DragObject* parameter to specify the drag cursor, or, optionally, a drag image list. There is no need to call the *Free* method for the *DragObject* when dragging is over.

If the *OnStartDrag* event handler sets the *DragObject* parameter to **nil**, a *TDragControlObject* object is automatically created and dragging begins on the control itself.

See also *DoStartDrag* method, *OnEndDrag* event, *TDragControlObject* object, *TDragObject* object

Methods

The following table lists the methods of *TControl*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Changed	protected	TControl
ChangeScale	protected	TControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Click	protected	TControl
ClientToScreen	public	TControl
Create	public	TControl
DblClick	protected	TControl
DefaultHandler	protected	TControl
DefineProperties	protected	TControl
Destroy	public	TControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
DoEndDrag	protected	TControl
DoStartDrag	protected	TControl
DragCanceled	protected	TControl
DragDrop	public	TControl
Dragging	public	TControl
DragOver	protected	TControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetClientOrigin	protected	TControl
GetClientRect	protected	TControl
GetDeviceContext	protected	TControl
GetDragImages	protected	TControl

Method	Visibility	Documented in
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetPalette	protected	TControl
GetParentComponent	protected	TControl
GetPopupMenu	protected	TControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HasParent	protected	TControl
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
Invalidate	public	TControl
MethodAddress	public	TObject
MethodName	public	TObject
MouseDown	protected	TControl
MouseMove	protected	TControl
MouseUp	protected	TControl
NewInstance	public	TObject
Notification	protected	TControl
PaletteChanged	protected	TControl
Perform	public	TControl
ReadState	protected	TControl
Refresh	public	TControl
RemoveComponent	public	TComponent
Repaint	public	TControl
SafeCallException	public	TComponent
ScreenToClient	public	TControl
SendCancelMode	protected	TControl
SendToBack	public	TControl
SetBounds	public	TControl
SetDragMode	protected	TControl
SetName	protected	TControl
SetParent	protected	TControl
SetParentComponent	protected	TControl
SetTextBuf	public	TControl
SetZOrder	protected	TControl
Show	public	TControl
Update	public	TControl

Method	Visibility	Documented in
UpdateBoundsRect	protected	TControl
VisibleChanging	protected	TControl
WndProc	protected	TControl

BeginDrag

BeginDrag starts the dragging of a control.

```
procedure BeginDrag(Immediate: Boolean);
```

Description Call *BeginDrag to* control the start of a drag operation. An application needs to call the *BeginDrag* method to begin dragging only when the *DragMode* property value for the control is *dmManual*.

If the *Immediate* parameter is *True*, the mouse pointer changes to the value of the *DragCursor* property and dragging begins immediately. If *Immediate* is *False*, the mouse pointer doesn't change to the value of the *DragCursor* property and dragging doesn't begin until the user moves the mouse pointer a short distance (5 pixels). This allows the control to accept mouse clicks without beginning a drag operation.

See also *DragCursor* property, *DragMode* property, *EndDrag* method

BringToFront

BringToFront puts the control in front of all other controls in its parent control.

```
procedure BringToFront;
```

Description *BringToFront* is especially useful for making certain that a form is visible. It is also used to reorder overlapping controls within a form.

The order in which controls stack on top of each other (also called the Z order) depends on the order the controls are placed on the form. For example, if you put a label and an image on a form so that one is on top of the other, the one that was placed first on the form becomes the one on the bottom. Because both the label and the image are non-windowed controls, they "stack" as you would expect them to. Call the *BringToFront* method for the bottom object to make it appear on top.

The stacking order of two windowed controls is the same as the stacking of two non-windowed controls. For example, if you put a memo on a form, then put a check box on top of it, the check box remains on top. Calling *BringToFront* for the memo moves the memo so that it appears on top.

The stacking order of windowed and non-windowed controls cannot be mingled. For example, if you put a memo, a windowed control, on a form, and then put a label, a non-windowed control, on top of it, the label disappears behind the memo. Windowed controls always stack on top of

non-windowed controls. In this example, calling the *BringToFront* method of the label does nothing, the label remains behind the memo.

See also *SendToBack* method

Changed

Changed sends a CM_CHANGED message to the control.

```
procedure Changed;
```

Description Call *Changed* after making changes to the properties of the control that affect the properties of a parent object. *Changed* sends a CM_CHANGED message back to the control. If the control is a windowed control, this message is bubbled up to the parent so that it can make any necessary adjustments.

See also *Parent* property

ChangeScale

ChangeScale repositions and resizes the control by a specified ratio.

```
procedure ChangeScale(M, D: Integer); dynamic;
```

Description Use *ChangeScale* to rescale a control. *ChangeScale* modifies the position as well as the size of a control. Thus, *ChangeScale* modifies the control's *Top*, *Left*, *Width*, and *Height* properties.

The *M* and *D* parameters define a fraction by which to scale the control. The *M* parameter is the multiplier and the *D* parameter is the divisor. For example, to make a control 75% of its original size, specify the value of *M* as 75, and the value of *D* as 100 (75/100). Alternately, the same results are achieved by specifying the value of *M* as 3, and the value of *D* as 4 (3/4). Both fractions are equal and result in the control being scaled by the same amount, 75%.

To scale the control to be 33% larger than its previous size, specify the value of *M* as 133, and the value of *D* as 100 (133/100). Alternately, specify the value of *M* as 4, and the value of *D* as 3 (4/3), as the fraction 133/100 is approximately equal to 4/3.

Controls derived from *TControl* might want to override *ChangeScale*. For example, a control which defines additional font properties should scale the heights of these fonts by the *M*/*D* ratio so that all the control's fonts scale uniformly. Similarly, any control which subdivides its client area into display regions and displays text in those regions might want to override *ChangeScale* to ensure those regions resize when the font does.

Note Because control sizes and coordinates are integers, scaling them by the *M*/*D* ratio incurs some degree of round-off error. Calling *ChangeScale* repeatedly means that at some point the control may disappear (when the round-off errors make the control smaller), or creep off the form (when the round-off errors move the location of control). When rescaling a control repeatedly,

consider destroying and reloading the form after several rescalings so that the coordinates are rescaled without most of the accumulating round-off error.

See also *Height* property, *Left* property, *Top* property, *Width* property

Click

Click calls the event handler attached to the control's *OnClick* event.

```
procedure Click; dynamic;
```

Description The *Click* method defined in *TControl* does nothing except call any event handler attached to the *OnClick* event. It is the protected implementation method for a control's *OnClick* event. Override *Click* to provide other responses in addition to the inherited event-handler call.

A control calls *Click* whenever it receives a left-button mouse-up message (*WM_LBUTTONUP*) if that message indicates that the mouse up occurred near the corresponding mouse-down. Many specific controls call *Click* in other circumstances, such as when a user types the shortcut key for a button or menu item.

See the *OnClick* event for a description of all occurrences that trigger click events.

See also *DblClick* method, *OnClick* event

ClientToScreen

ClientToScreen translates the given point from client area coordinates to global screen coordinates.

```
TPoint = record
  X: Longint;
  Y: Longint;
end;

function ClientToScreen(const Point: TPoint): TPoint;
```

Description Use *ClientToScreen* to convert a point whose coordinates are expressed locally to the control to the corresponding point in screen coordinates. In client area coordinates (0, 0) corresponds to the upper left corner of the control's client area. In screen coordinates (0, 0) corresponds to the upper left corner of the screen.

Use *ScreenToClient* along with *ClientToScreen* to convert from one control's coordinate system to another control's coordinate system. For example,

```
P := TargetControl.ScreenToClient(SourceControl.ClientToScreen(P));
```

converts *P* from coordinates in *SourceControl* to coordinates in *TargetControl*.

See also *ScreenToClient* method

Create

Create creates an instance of *TControl*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to construct and initialize an instance of *TControl*.

After calling the inherited method, Create initializes the following properties:

- *Color* to *clWindow*.
- *ControlStyle* to [*csCaptureMouse, csClickEvents, csSetCaption, csDoubleClick*].
- *DragCursor* to *crDrag*.
- *Enabled* and *Visible* to *True*.
- *IsControl* to *False*.
- *ParentColor, ParentFont,* and *ParentShowHint* to *True*.

When overriding *Create*, always call the inherited *Create* method first, then proceed with the control's initialization. Remember to specify the override directive when overriding the *Create* method.

If a component's *Create* method allocates resources or memory, also override the *Destroy* method to free those resources.

See also *Destroy* method, *TComponent.Create* method

DblClick

DblClick calls the event handler attached to the control's *OnClick* event.

```
procedure DblClick; dynamic;
```

Description The *DblClick* method as implemented in *TControl* does nothing except call any event handler attached to the *OnDblClick* event. It is the protected implementation method for a control's *OnDblClick* event. Override *DblClick* to provide other responses in addition to the inherited event-handler call.

The *DblClick* method is triggered by a left-button double-click message (*WM_LBUTTONDBLCLICK*) from Windows.

See also *Click* method, *OnDblClick* event

DefaultHandler

DefaultHandler provides message handling for all messages that a control does not have specific handlers for.

```
procedure DefaultHandler(var Message); override;
```

Description Override *DefaultHandler* to extend the default handling of messages. *DefaultHandler* is the final opportunity for component writers to handle messages to controls. The *WndProc* method provides the first opportunity. Individual message methods such as *Click* provide additional opportunities.

DefaultHandler handles the messages Windows sends to manage the control's text: *WM_GETTEXT*, *WM_SETTEXT*, and *WM_GETTEXTLENGTH*.

See also *TObject.DefaultHander* method, *WndProc* method

DefineProperties

DefineProperties makes the *IsControl* protected property persistent, providing methods to read and write *IsControl* to a stream, such as a form file.

```
procedure DefineProperties(Filer: TFiler); override;
```

Description Override *DefineProperties* to define new properties or publish unpublished properties. To define a new property, call the *DefineBinaryProperty* or *DefineProperty* method of the *TFiler* object passed as the *Filer* parameter.

Note When overriding this method, call the inherited *DefineProperties* method first.

See also *TFiler.DefineBinaryProperty* method, *TFiler.DefineProperty* method, *TPersistent.DefineProperties* method

Destroy

Destroy disposes of an instance of TControl.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the control is not already freed, and only then calls *Destroy*.

Applications should only free controls directly when the *Create* method was called to create an instance of the control without giving the control an owner.

Override *Destroy* to free any memory or resources allocated in the *Create* method. When declaring a *Destroy* method in a descendent control type, always add the **override** directive to the declaration and call the **inherited** *Destroy* as the last statement in the redeclared method.

When a control is created, Delphi automatically provides exception handling around the constructor call. The destructor is called if an exception escapes from the constructor. This means that the destructor code must be prepared to clean up a partially constructed instance. Check all data fields for zero before disposing of their contents.

See also *Create* method, *Free* method, *Owner* property

DoEndDrag

DoEndDrag is the protected implementation method for a control's *OnEndDrag* event.

```
procedure DoEndDrag(Target:TObject); X, Y: Integer); dynamic;
```

Description The *DoEndDrag* method does nothing except call any event handler attached to the *OnEndDrag* event. When creating a component that descends from *TControl*, override *DoEndDrag* to provide other responses in addition to the inherited event-handler call.

See also *DoStartDrag* method, *OnEndDrag* event

DoStartDrag

DoStartDrag is the protected implementation method for a control's *OnStartDrag* event.

```
procedure DoStartDrag(var DragObject: TDragObject); dynamic;
```

Description The *DoStartDrag* method does nothing except call any event handler attached to the *OnStartDrag* event. When creating a component that descends from *TControl*, override *DoStartDrag* to provide other responses in addition to the inherited event-handler call.

See also *DoEndDrag* method, *OnStartDrag* event

DragCanceled

DragCanceled does nothing except provide an interface for a method to respond to the cancellation of a drag.

```
procedure DragCanceled; dynamic;
```

Description When creating a component that descends from *TControl*, override *DragCanceled* to provide code that executes when a drag operation is canceled.

See also *DoEndDrag* method, *DoStartDrag* method, *DragDrop* method, *Dragging* method

C

DragDrop

DragDrop calls the *OnDragDrop* event handler, if one exists.

```
procedure DragDrop(Source: TObject; X, Y: Integer);
```

Description When creating a component that descends from *TControl*, override *DragDrop* to add additional code that executes before the *OnDragDrop* event handler is called.

The *Source* parameter of the *DragDrop* method is the object to be considered dropped by the handler. The *X* and *Y* parameters are the coordinates of the mouse positioned where the object is to be considered dropped.

See also *DragCanceled* method, *OnDragDrop* event

Dragging

Dragging specifies whether a control is being dragged.

```
function Dragging: Boolean;
```

Description Use *Dragging* to determine the drag state of the control. If *Dragging* returns *True*, the control is being dragged. If *Dragging* is *False*, the control is not being dragged.

See also *DragCursor* property, *DragMode* property, *OnDragDrop* event, *OnDragOver* event

DragOver

DragOver calls the *OnDragOver* event handler, if one exists.

```
TDragState = (dsDragEnter, dsDragLeave, dsDragMove);

procedure DragOver(Source: TObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean); dynamic;
```

Description Override *DragOver* to add additional code that executes before the *OnDragOver* event handler is called.

Set the *Accept* parameter to *True*, to indicate that the user can drop the dragged object on the control. Set *Accept* to *False*, to indicate that the user can't drop the dragged object on the control.

The *Source* parameter is the object being dragged. These are the possible values of the *State* parameter:

Value	Meaning
dsDragEnter	The mouse is entering the control.
dsDragMove	The mouse moved while over the control.
dsDragLeave	The mouse is leaving the control.

See also *DragDrop* method, *OnDragOver* event

EndDrag

EndDrag stops a control from being dragged any further.

```
procedure EndDrag(Drop: Boolean);
```

Description Use *EndDrag* to stop a dragging operation that began with a call to the *BeginDrag* method.

If the *Drop* parameter is *True*, the control being dragged is dropped. If the *Drop* parameter is *False*, the control is not dropped and dragging is canceled.

See also *BeginDrag* method, *OnEndDrag* event

GetClientOrigin

GetClientOrigin obtains the screen coordinates of the top-left corner of the control's client area.

```
function GetClientOrigin: TPoint; virtual;
```

Description Override *GetClientOrigin* to change how the *ClientOrigin* is calculated.

The *GetClientOrigin* method is a protected method for the *ClientOrigin* property. *GetClientOrigin* calculates the *ClientOrigin* by adding the control's Left and Top values to the *ClientOrigin* coordinates of its parent.

See also *ClientOrigin* property, *GetClientRect* method, *Left* property, *Top* property

GetClientRect

GetClientRect returns a rectangle defining the client area of the control.

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;

function GetClientRect: TRect; virtual;
```

Description Override *GetClientRect* to change how the client area is calculated.

The *GetClientRect* method is a protected method for the *ClientRect* property. *GetClientRect* returns a rectangle with the rectangle's *Top* and *Left* fields set to zero, and its *Bottom* and *Right* fields set to the control's *ClientHeight* and *ClientWidth*, respectively.

See also *BoundsRect* property, *ClientHeight* property, *ClientWidth* property, *GetClientOrigin* method

C

GetDeviceContext

GetDeviceContext returns a device context for the control.

```
function GetDeviceContext(var WindowHandle: HWnd): HDC; virtual;
```

Description Call *GetDeviceContext* to obtain a device context. Override *GetDeviceContext* to change how the device context is obtained.

Pass the window handle of the control in the *WindowHandle* parameter. *GetDeviceContext* returns the device context returned by its parent control's *GetDeviceContext* method.

See also *Parent* property, *TWinControl.WindowHandle* property

GetDragImages

GetDragImages returns the control's image list containing the image to be displayed while dragging.

```
function GetDragImages: TCustomImageList; virtual;
```

Description Override *GetDragImages* to provide a custom image list for the control. The *GetDragImages* method for *TControl* returns **nil**.

See also *TCustomImageList* object

GetPalette

The *GetPalette* method returns the handle to a palette for use by the control.

```
function GetPalette: HPALETTE; dynamic;
```

Description The *GetPalette* for *TControl* returns a null handle, indicating that the control does not have a palette. Override *GetPalette* to implement derived objects that use a palette. Create a palette handle for the control by a call to the Windows API function *CreatePalette*. *GetPalette* should return the palette handle for the control.

Specifying the palette for a control tells the application that the control's palette needs to be realized and designates the palette to use for realization.

GetPalette is called only if the runtime video mode requires color palette support, for example, in 256-color mode.

See also *PaletteChanged* method

GetParentComponent

GetParentComponent returns the parent component of the control.

```
function GetParentComponent: TComponent; override;
```

Description Call *GetParentComponent* to find the parent component of the control. *GetParentComponent* is a protected access method of the control's *Parent* property.

See also *Parent* property

GetPopupMenu

GetPopupMenu is the pop-up menu associated with the control.

```
function GetPopupMenu: TPopupMenu; dynamic;
```

Description Call *GetPopupMenu* to find the popup menu associated with the control. Override *GetPopupMenu* to change which popup menu is associated with the control or to specify different popup menus depending on the state of the control.

GetPopupMenu returns the value of the *Popup* property.

See also *PopupMenu* property

GetTextBuf

GetTextBuf retrieves the control's text and copies it into a buffer and returns the number of characters copied.

```
function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;
```

Description Call *GetTextBuf* to retrieve the text of a control into a fixed size buffer. The text copied is the value of the *Text* property. *GetTextBuf* returns the number of characters that were actually copied, which is either the length of the *Text* property, or *BufSize* – 1, whichever is smaller.

To find out how many characters the buffer needs to hold the entire text, call the *GetTextLen* method before allocating a buffer for *GetTextBuf*.

GetTextBuf is available for compatibility with 16-bit code. Where backward compatibility is not an issue, use the *Text* property.

See also *GetTextLen* method, *Text* property

GetTextLen

GetTextLen returns the length of the control's text.

```
function GetTextLen: Integer;
```

Description Call *GetTextLen* to find the length of the control's text. This is the size needed for a text buffer to be used by the *GetTextBuf* method.

See also *GetTextBuf* method, *Text* property

C

HasParent

HasParent indicates whether the control has a parent.

```
function HasParent: Boolean; override;
```

Description Call *HasParent* to find out if a control has a parent.

The parent of the control is responsible for writing the control to a stream. By default, *HasParent* for a control returns *True*. The parent that writes the control to a stream is usually a form or panel control that contains this control.

Note A child control that returns *True* from *HasParent* must implement the *GetParentComponent* and *SetParentComponent* methods.

See also *GetParentComponent* method, *SetParentComponent* method

Hide

Hide makes the control invisible.

```
procedure Hide;
```

Description Call *Hide* to hide a control. *Hide* sets the *Visible* property of the control to *False*.

Although a control that is hidden is not visible, its properties and methods are still available.

See also *Show* method, *Visible* property

Invalidate

Invalidate tells Windows to repaint the control after other important Windows messages are handled.

```
procedure Invalidate; virtual;
```

Description Use *Invalidate* when the entire control needs to be repainted. When more than one region within the control needs repainting, *Invalidate* will cause the entire window to be repainted in a single pass, avoiding flicker caused by redundant repaints. There is no performance penalty for calling *Invalidate* multiple times before the control is actually repainted.

See also *Refresh* method, *Repaint* method, *Update* method

MouseDown

MouseDown calls the *OnMouseDown* event handler, if one exists.

```
type
  TMouseButton = (mbLeft, mbRight, mbMiddle);
  TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); dynamic;
```

Description Override the protected *MouseDown* method to provide other responses in addition to calling the *OnMouseDown* event handler.

The *Button* parameter determines which mouse button the user pressed. Shift indicates which shift keys (*Shift, Ctrl,* or *Alt*) were down when the user pressed or released the mouse button that generated the mouse-button event. *X* and *Y* are the pixel coordinates of the mouse pointer within the client area of the control.

A control calls *MouseDown* in response to any of the Windows mouse-down messages (*WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN*), decoding the message parameters into the shift-key state and position, which it passes in the *Shift, X,* and *Y* parameters, respectively. The value of the *Button* parameter depends on which of the Windows messages triggered the event.

See also *MouseMove* event, *MouseUp* event, *OnMouseDown* event

MouseMove

MouseMove calls the event handler attached to the *OnMouseMove* event.

```
type TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

procedure MouseMove(Shift: TShiftState; X, Y: Integer); dynamic;
```

Description Override the protected *MouseMove* method to provide other responses in addition to calling the *OnMouseMove* event handler.

A control calls *MouseMove* in response to any of the Windows mouse-move messages (*WM_MOUSEMOVE*), decoding the message parameters into the shift-key state and position, which it passes in the *Shift*, *X*, and *Y* parameters, respectively.

See also *MouseDown* method, *MouseUp* method, *OnMouseMove* event

MouseUp

MouseUp calls the *OnMouseUp* event handler.

```
type
  TMouseButton = (mbLeft, mbRight, mbMiddle);
  TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); dynamic;
```

Description Override the protected *MouseUp* method to provide other responses in addition to calling the *OnMouseUp* event handler.

A control calls *MouseUp* in response to any of the Windows mouse-up messages (*WM_LBUTTONUP*, *WM_MBUTTONUP*, *WM_RBUTTONUP*), decoding the message parameters into the shift-key state and position, which it passes in the *Shift*, *X*, and *Y* parameters, respectively. The value of the *Button* parameter depends on which of the Windows messages triggered the event.

See also *MouseDown* method, *MouseMove* method, *OnMouseUp* event

Notification

Notification allows *TControl* to update its *Popupmenu* property when the pop-up menu it refers to is destroyed.

```
type
  TOperation = (opInsert, opRemove);

procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description Override the *Notification* method to respond to notifications that other components are about to be destroyed or have just been created. Update

controls that rely on other system resources so that they will not contain references to objects that have been removed. For example, data-aware objects should override the *Notification* method to update themselves when their data source is removed.

By default, components pass along the notification to their owned components, if any.

See also *PopupMenu* property, *TComponent.Notification* property

PaletteChanged

PaletteChanged responds to changes in the system's palette by realizing the control's palette.

```
function PaletteChanged(Foreground: Boolean): Boolean; dynamic;
```

Description Override *PaletteChanged* to change how the control responds to Windows notifications that the system palette is changing. *TControl* responds to these notifications by trying to realize the control's palette, if any, into the current device context. If the change in the system palette causes the actual value of the control's palette to change, *PaletteChanged* will invalidate the control so that it can repaint with the new palette.

Windows paints the active window with a foreground palette, while other windows are forced to use background palettes. Background palettes are approximate matches to the colors specified by the control's logical palette, given the limitations imposed by implementing the foreground palette. Windows only allows a single foreground palette.

PaletteChanged allows the control to obtain a new realization of its palette when the system palette changes. When *ForeGround* is *True*, the form has been activated and the control is specifying the new foreground palette. When *ForeGround* is *False*, another application or form has changed the foreground palette, and controls that are sensitive to the available palette should realize new background palettes to best match their logical palettes.

If the control does not have a logical palette to be realized into the current device context, *GetPalette* will return a handle of 0, and *PaletteChanged* will return *False*, indicating that no palette was realized. If *GetPalette* returns the handle to a logical palette, *PaletteChanged* will realize this palette into the current device context, and return *True*.

Note *PaletteChanged* is called only when the runtime video mode requires palette support, such as for 256-color mode, but not 16 million color mode.

See also *GetDeviceContext* method, *GetPalette* method

Perform

Perform responds as if the control received the specified Windows message.

```
function Perform(Msg: Cardinal; WParam, LParam: Longint): Longint;
```

Description Call *Perform* to bypass the Windows message queue and send a message directly to the control's window procedure.

Perform fills a message record (of type *TMessage*) with the message ID passed in the *Msg* parameter, the message parameters passed in *WParam* and *LParam*, and a result field of zero. *Perform* then passes the message record to the *WndProc* method for processing.

See also *TMessage* type

ReadState

ReadState prepares the control for having its properties assigned values read from a stream.

```
procedure ReadState(Reader: TReader); override;
```

Description Override *ReadState* to change the preparations the control makes as it readies itself to have the values of its properties assigned from a stream. For example, a control might destroy temporary internal data structures or objects before new instances are loaded from the stream. Remember to include a call to the inherited method when overriding *ReadState*.

TControl.ReadState assigns the *TReader* object's *Parent* property to this control's own *Parent* property before calling the inherited *ReadState* method of *TComponent*. Finally, *ReadState* synchronizes the *Font*, *Color*, and *ShowHint* properties with the parent control, as indicated by the *ParentFont*, *ParentColor*, and *ParentShowHint* properties, respectively.

See also *Parent* property, *ParentColor* property, *ParentFont* property, *ParentShowHint* property, *TComponent.ReadState* method, *TReader* object

Refresh

Refresh repaints the control on the screen.

```
procedure Refresh;
```

Description Call *Refresh* method to repaint the control immediately. *Refresh* calls the *Repaint* method. Use the *Refresh* and *Repaint* methods interchangeably.

See also *Invalidate* method, *Parent* property, *Repaint* method, *Update* method

Repaint

Repaint forces the control to repaint its image on the screen.

```
procedure Repaint;
```

Description Call *Repaint* to force the control to repaint its image immediately. If the *ControlStyle* property includes *csOpaque*, the control paints itself directly. Otherwise, the *Repaint* method calls the *Invalidate* method and then the

Update method so that any visible portions of controls beneath the control will be repainted as well.

See also *ControlStyle* property, *GetDeviceContext* method, *Invalidate* method, *Parent* property, *Update* method

ScreenToClient

ScreenToClient converts the screen coordinates of a specified point on the screen to client coordinates.

```
TPoint = record
  X: Longint;
  Y: Longint;
end;

function ScreenToClient(const Point: TPoint): TPoint;
```

Description Use *ScreenToClient* to convert a point in screen coordinates to local, or client area, coordinates. In client area coordinates (0, 0) corresponds to the upper left corner of the control's client area. In screen coordinates (0, 0) corresponds to the upper left corner of the screen.

Use *ScreenToClient* along with *ClientToScreen* to convert from one control's coordinate system to another control's coordinate system. For example,

```
P := TargetControl.ScreenToClient(SourceControl.ClientToScreen(P));
```

converts *P* from coordinates in *SourceControl* to coordinates in *TargetControl*.

See also *ClientToScreen* method

SendCancelMode

SendCancelMode cancels the modal state of the control.

```
procedure SendCancelMode(Sender: TControl);
```

Description Call *SendCancelMode* to cancel the modal state of the control. Various controls implemented in the VCL support a model state where the user must respond to the control before interacting with any other object on the form. An example would be the dropped-down state of a combo box when the drop-down list is showing. For these controls, specific user actions send a windows message to the control to end the model state, such as a mouse click that closes up the dropped-down list.

SendCancelMode allows an application to end the modal state of the control in the absence of a user action, or in order to implement a new user interface response.

See also *Perform* method, *TCustomComboBox* object

SendToBack

SendToBack puts a windowed control behind all other windowed controls within its parent component or form, or it puts a non-windowed control behind all other non-windowed control within its parent component or form.

```
procedure SendToBack;
```

Description *Use SendToBack* to change the order of overlapping controls or forms.

The order in which controls stack on top of each other (also called the Z order) depends on the order the controls are placed on the form. For example, if you put a label and an image on a form so that one is on top of the other, the one that was placed first on the form becomes the one on the bottom. Because both the label and the image are non-windowed controls, they "stack" as you would expect them to. Call the *SendToBack* method for the top object to move it below the other object.

The stacking order of two windowed controls is the same as the stacking of two non-windowed controls. For example, if you put a memo on a form, then put a check box on top of it, the check box remains on top. Calling *SendToBack* for the check box makes the memo appear on top.

The stacking order of windowed and non-windowed controls cannot be mingled. For example, if you put a memo, a windowed control, on a form, and then put a label, a non-windowed control, on top of it, the label disappears behind the memo. Windowed controls always stack on top of non-windowed controls. In this example, calling the *SendToBack* method of the memo does nothing, the label remains behind the memo.

If the control has the input focus when the *SendToBack* method executes, it loses the input focus.

See also *BringToFront* method

C

SetBounds

SetBounds sets the *Left*, *Top*, *Width*, and *Height* properties all at once.

```
procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer); virtual;
```

Description *Use SetBounds* to change all of the component's boundary properties at one time. The same effect can be achieved by setting the *Left*, *Top*, *Width*, and *Height* properties separately, but *SetBounds* changes all four properties at once ensuring that the control will not repaint between changes.

Specify the values for the *Left*, *Top*, *Width*, and *Height* properties as the value of the *ALeft*, *ATop*, *AWidth*, and *AHeight* parameters, respectively.

See also *Height* property, *Left* property, *Top* property, *Width* property

SetDragMode

SetDragMode sets the drag mode for the control.

```
type TDragMode = (dmManual, dmAutomatic);
procedure SetDragMode(Value: TDragMode); virtual;
```

Description *SetDragMode* is the protected method for the *DragMode* property. Override *SetDragMode* to execute additional code when the value of the *DragMode* property changes.

See also *DragMode* property

SetName

SetName sets the name of the control.

```
type TComponentName = type string;
procedure SetName(const Value: TComponentName); override;
```

Description *SetName* is the protected method for setting the value of the *Name* property. Override *SetName* to execute additional code when the value of the *Name* property changes.

See also *Name* property

SetParent

SetParent sets the parent of the control

```
procedure SetParent(AParent: TWinControl); virtual;
```

Description Call *SetParent* to make the specified window control the parent of this control. The *SetParent* method is a protected method for setting the *Parent* property. Override *SetParent* to execute additional code when the value of the *Parent* property changes.

Specify the control that is to become the parent control as the value of the *AParent* parameter.

If the control already has a parent, *SetParent* removes the control from that parent's list of controls by calling the parent's *RemoveControl* method. If *AParent* is non-**nil**, *SetParent* calls that control's *InsertControl* method to add the control to its list of controls.

The *Destroy* method of all controls calls *SetParent*(**nil**) to remove the control from its parent's control list before destroying the component.

See also *Destroy* method, *HasParent* method, *InsertControl* method, *Parent* property, *RemoveControl* method

SetParentComponent

SetParentComponent the parent for the control.

```
procedure SetParentComponent(Value: TComponent); override;
```

Description *SetParentComponent* is used by the streaming system to make the specified component the parent of the control. The *Value* parameter is the component that is to become the parent of the control.

SetParentComponent calls the *SetParent* method only if the component specified as the parent is a windowed control.

See also *HasParent* method, *SetParent* method

SetTextBuf

SetTextBuf sets text of the control.

```
procedure SetTextBuf(Buffer: PChar);
```

Description Call the *SetTextBuf* method to set the text of the control to the text in the specified buffer.

The text is set to the text in the buffer pointed to by the *Buffer* parameter. *Buffer* must point to a null-terminated string. The *SetTextBuf* method sends the *WM_SETTEXT* and *CM_TEXTCHANGED* messages.

Note The *SetTextBuf* method is provided for backwards compatibility only. Use the *Text* property instead to set the text of the control.

See also *Text* property

SetZOrder

SetZOrder repositions the control in its parent's control list, thereby changing position onscreen also.

```
procedure SetZOrder(TopMost: Boolean); dynamic;
```

Description Call *SetZOrder* to change the Z-order of the control. To make the control the topmost control, specify the *TopMost* value as *True*. To make the control the bottommost, set *TopMost* to *False*.

After changing the order, *SetZOrder* invalidates the control to ensure repainting to reflect the new order.

The stacking order of windowed and non-windowed controls cannot be mingled. For example, if you put a memo, a windowed control, on a form, and then put a label, a non-windowed control, on top of it, the label disappears behind the memo. Windowed controls always stack on top of non-windowed controls. In this example, calling the *SetZOrder* method of the label or the memo has no effect, the label always remains behind the memo.

See also *BringToFront* method, *Invalidate* method, *SendToBack* method

Show

Show makes a control visible.

```
procedure Show;
```

Description Use the *Show* method to make visible a control that was previously hidden.

The *Show* method sets the control's *Visible* property to *True* and also ensures that the parent control is visible.

See also *Hide* method, *Visible* property

Update

Update processes any pending paint messages immediately.

```
procedure Update; virtual;
```

Description Call *Update* to force the control to be repainted before any more, possibly time-consuming, processing takes place. Use *Update* to provide immediate feedback to the user that cannot wait for the Windows paint message to arrive. *Update* does not invalidate the control, but simply forces a repaint of any regions that have already been invalidated. Call *Repaint* instead if you need to invalidate the control as well.

See also *Invalidate* method, *Refresh* method, *Repaint* method

UpdateBoundsRect

UpdateBoundsRect changes the *Top, Left, Width,* and *Height* properties of the control, but doesn't update its screen image.

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;

procedure UpdateBoundsRect(const R: TRect);
```

Description *UpdateBoundsRect* is used internally by controls to stay synchronized with changes made by Windows. *UpdateBoundsRect* updates the properties to match those passed in the rectangle as the *R* parameter.

See also *BoundsRect* method

VisibleChanging

VisibleChanging is called before the value of a control's *Visible* property changes.

```
procedure VisibleChanging; dynamic;
```

Description *VisibleChanging* does nothing more than provide an interface for descendent objects to override in order to respond to changes in the control's visibility. For example, if the control should not change its visibility, override *VisibleChanging* to raise an exception when an attempt is made to alter the control's visibility.

See also *Visible* property

WndProc

WndProc provides specific message responses for the control.

```
procedure WndProc(var Message: TMessage); virtual;
```

Description Override *WndProc* to change the initial Windows message handler for the control. The *WindowProc* property is initialized to *WndProc*.

WndProc implements the mouse behavior specified by the *ControlStyle* and *DragMode* properties, and updates the *ControlState* property. It passes any unhandled messages on to the *Dispatch* method.

When overriding *WndProc* to provide specialized responses to messages, be sure to call the inherited *WndProc* at the end to dispatch any unhandled messages.

See also *ControlState* property, *ControlStyle* property, *DragMode* **property**, *TObject.Dispatch* method, *WindowProc* method

TCoolBar

Unit: ComCtrls Palette: Win32

TObject • TPersistent • TComponent • TControl • TWinControl • TToolWindow • TCoolBar

TCoolBar is a container for *TCoolBand* objects. It displays a collection of windowed controls within movable, resizable bands.

Description A *TCoolBar* contains child controls that can be moved and resized independently. Each control resides on an individual *band*, represented by a *TCoolBand* object listed in the *Bands* property. The user positions the controls by dragging the sizing grip to the left of each band.

A *TCoolBar* can contain objects that do not descend from *TWinControl*, such as *TLabel* or *TSpeedButton*. Only *TWinControl* descendants, however, appear on resizable bands.

Note *TCoolBar* requires version 4.70 or later of COMCTL32.DLL. This file, which usually resides in the WINDOWS\SYSTEM or WINDOWS\SYSTEM32 directory, is required at both design time and runtime. When deploying applications that use *TCoolBar*, be sure to install COMCTL32.DLL.

The component represented by *TCoolBar* is sometimes referred to as a *rebar*.

See also *TCoolBand* object, *TCoolBands* object

Properties

The following table lists the properties of *TCoolBar*:

Property	Visibility	Default	Documented in
Align	published	alTop	TCoolBar
AutoSize	published	False	TCoolBar
BandBorderStyle	published	bsSingle	TCoolBar
Bands	published		TCoolBar
Bitmap	published		TCoolBar
BorderWidth	published	0	TToolWindow
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	public		TWinControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
EdgeBorders	published	[ebLeft, ebTop, ebRight, ebBottom]	TToolWindow
EdgeInner	published	edRaised	TToolWindow
EdgeOuter	published	esLowered	TToolWindow
FixedOrder	published	False	TCoolBar
FixedSize	published	False	TCoolBar
Font	published		TControl

Property	Visibility	Default	Documented in
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Images	published		TCoolBar
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	True	TControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ShowHint	published		TControl
Showing	public, read-only		TWinControl
ShowText	published	True	TCoolBar
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Vertical	published	False	TCoolBar
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

C

Align

Determines the alignment of the *TCoolBar* component.

property Align: TAlign;

Description Setting *Align* to *alLeft* or *alRight* automatically sets *Vertical* to *True*. Setting *Align* to *alTop*, *alBottom*, *alClient*, or *alNone* automatically sets *Vertical* to *False*. In other respects, *Align* behaves like the inherited property.

See also *TAlign* type, *TControl.Align* property, *Vertical* property

AutoSize

Determines whether the *TCoolBar* resizes automatically to accommodate the bands it contains.

property AutoSize: Boolean;

Description When *AutoSize* is *True*, the *TCoolBar* component changes its width or height to accommodate the size and position of the bands within it. The *Vertical* property determines which dimension is adjusted: If *Vertical* is *True*, the component's width changes. If *Vertical* is *False*, the component's height changes.

See also *Align* property, *Height* property, *Vertical* property, *Width* property

BandBorderStyle

Determines the border style of the bands in the *TCoolBar*.

```
property BandBorderStyle: TBorderStyle;
```

Description If *BandBorderStyle* is *bsSingle*, each band in the *TCoolBar* has a single line around it. If *BandBorderStyle* is *bsNone*, the bands have no borders.

Bands

A collection of bands (*TCoolBands*).

```
property Bands: TCoolBands;
```

Description The *Bands* property holds a *TCoolBands*—that is, a collection of *TCoolBand* objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the *Bands* property in the Object Inspector, then double-click in the Value column to the right or click the ellipsis (...) button.

You can also create bands simply by adding new windowed controls from the palette. When any *TWinControl* descendant is placed on the form as a child of the *TCoolBar*, a new band is created for it.

See also *TCoolBand* object, *TCoolBands* object

Bitmap

The image that appears behind the bands in the *TCoolBar*.

```
property Bitmap: TBitmap;
```

Description *Bitmap* holds a *TBitmap* object. It determines the image that appears behind the bands in the *TCoolBar*.

By default, the *Bitmap* image is positioned relative to the upper left corner of the *TCoolBar* and covers the client area, where the transparent bands allow it to display without occlusion. This appearance can be modified by setting the *FixedBackground*, *Bitmap*, and *ParentBitmap* properties of *TCoolBand*.

See also *TBitmap* object, *TCoolBand.Bitmap* property, *TCoolBand.FixedBackground* property, *TCoolBand.ParentBitmap* property

FixedOrder

Determines whether the user can reorder the bands in the *TCoolBar*.

property FixedOrder: Boolean;

Description If *FixedOrder* is set to *True*, the user can reposition the bands by dragging them with the mouse, but cannot change their order. Bands can still be reordered from the Collection editor at design time, or by changing their *Index* properties in program code.

See also *TCoolBand.Index* property

FixedSize

Determines whether the bands automatically maintain a uniform height (or width).

property FixedHeight: Boolean;

Description Ordinarily, when *Vertical* and *FixedSize* are both *False*, the height of each *TCoolBand* is determined by the height of the *TWinControl* component linked to it. (If there is no *TWinControl* component linked to a band, its height is determined by the text it contains.) When *FixedSize* is set to *True*, the bands are resized so that they all have the same height; increasing the height of the tallest visible band automatically increases the height of all the other bands.

When *Vertical* is *True*, these dependencies are switched. As long as *FixedSize* is *False*, the width of each *TCoolBand* is determined by the width of the *TWinControl* component linked to it. If *FixedSize* and *Vertical* are both set to *True*, the bands maintain the same width; increasing the width of the widest visible band automatically increases the width of all the other bands.

See also *TCoolBand* object, *TCoolBand.Text* property, *Vertical* property

C

Images

A list of images available for bands in the *TCoolBar*.

property Images: TImageList;

Description *Images* is a list of bitmaps that can be displayed to the left of any band in the *TCoolBar*.

See also *TImageList* object, *TCoolBand.ImageIndex* property

ShowText

Determines whether text is displayed on the bands.

property ShowText: Boolean;

Description If *ShowText* is *True*, the *Text* string of each *TCoolBand* is displayed on the band at runtime. If *ShowText* is *False*, only the *TWinControl* linked to the band is displayed.

See also *TCoolBand.Text* property

Vertical

Determines whether the bands are arranged vertically or horizontally.

property Vertical: Boolean;

Description If *Vertical* is *False* (the default), the bands in the *TCoolBar* are oriented horizontally, from left to right and top to bottom. If *Vertical* is *True*, the bands are oriented vertically, from top to bottom and left to right.

See also *Align* property, *AutoSize* property, *FixedSize* property

Events

The following table lists the events of *TCoolBar*:

Event	Visibility	Documented in
OnChange	published	TCoolBar
OnClick	published	TControl
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnResize	published	TCoolBar
OnStartDrag	published	TControl

OnChange

OnChange occurs when the position or size of a band changes.

property OnChange: TNotifyEvent;

Description The *OnChange* event occurs whenever a *TCoolBand*'s *Break*, *Index*, or *Width* property is modified at runtime. An *OnChange* event handler can be used, for example, to record the positions and sizes of the bands in the Windows registry.

See also *TCoolBand.Break* property, *TCoolBand.Index* property, *TCoolBand.Width* property, *TNotifyEvent* type

OnResize

OnResize occurs when the *TCoolBar* is resized.

property OnResize: TNotifyEvent;

Description The *OnResize* event occurs whenever the entire *TCoolBar* is resized at runtime.

See also *TNotifyEvent* type

Methods

The following table lists the methods of *TCoolBar*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCoolBar
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCoolBar
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl

Method	Visibility	Documented in
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl

Method	Visibility	Documented in
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Creates and initializes a *TCoolBar* instance.

constructor Create(AOwner: TComponent); **override**;

Description *Create* generates a *TCoolBar* object and initializes its properties. If COMCTL32.DLL cannot be found, *Create* raises the *EComponentError* exception.

See also *EComponentError* object

Destroy

Destroys the *TCoolBar* instance and frees its memory.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the control is not already destroyed, and only then calls *Destroy*.

See also *Free* method

TCurrencyField

Unit: DB

TObject • TPersistent • TComponent • TField • TNumericField • TFloatField • TCurrencyField

A *TCurrencyField* object represents a field that contains currency values in a dataset.

Description *Currency* fields can hold values in the range from (positive or negative) 5.0 * 10 – 324 to 1.7 * 10308 with an accuracy of 15 digits.

TCurrencyField differs from its immediate ancestor *TFloatField* only in having a *DataType* of *ftCurrency*, and in formatting the value using a currency format by default. As a descendant of *TFloatField*, *TCurrencyField* includes many properties, methods, and events that are useful for managing the value and properties of a floating-point field in a database.

Do not confuse *TCurrencyField* with the *Currency* data type. Currency fields use the *Double* data type to store and manipulate their values. This data type is the format used by the physical database tables for currency fields. The *TBCDField* class uses the *Currency* data type to store and manipulate its value.

See also *TBCDField* object

Properties

The following table lists the properties of *TCurrencyField*:

Property	Visibility	Default	Documented in
Alignment	published	taRightJustify	TField
AsBoolean	public		TField
AsCurrency	public		TField
AsDateTime	public		TField
AsFloat	public		TFloatField
AsInteger	public		TFloatField
AsString	public		TFloatField
AsVariant	public		TFloatField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
Currency	published	True	TFloatField
CurValue	public, read-only		TField
CustomConstraint	published		TField
DataSet	public		TField
DataSize	public, read-only		TFloatField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayFormat	published		TNumericField
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditFormat	published		TNumericField
EditMask	public		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField

Property	Visibility	Default	Documented in
FieldNo	public, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField
MaxValue	published		TFloatField
MinValue	published		TFloatField
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
Precision	published	15	TFloatField
ReadOnly	published	False	TField
Required	published	False	TField
Size	public		TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TFloatField
VCLComObject	public		TComponent
Visible	published	True	TField

Events

The following table lists the events of *TCurrencyField*:

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TCurrencyField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TCurrencyField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject

Method	Visibility	Documented in
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create creates an instance of *TCurrencyField*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TCurrencyField*. After calling the inherited constructor, *Create* sets the *DataType* to *ftCurrency*, and the *Currency* property to *True*.

It is seldom necessary to call *Create* directly, because a currency field object is instantiated automatically for all currency fields in a dataset.

See also *Currency* property, *DataType* property, *TNumericField.Create* method

TCustomCheckBox

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TButtonControl • TCustomCheckBox

TCustomCheckBox is the ancestor of all check-box components.

Description *TCustomCheckBox* is an abstract class from which check-box components, including *TCheckBox* and *TDBCheckBox*, descend. Check boxes present the user with options that can be selected (checked) or deselected (unchecked).

See also *TCheckBox* object, *TDBCheckBox* object

Properties

The following table lists the properties of *TCustomCheckBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
Alignment	protected	taRightJustify	TCustomCheckBox
AllowGrayed	protected	False	TCustomCheckBox
BoundsRect	public		TControl
Brush	public, read-only		TWinControl

Property	Visibility	Default	Documented in
Checked	protected		TCustomCheckBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
State	protected	cbUnchecked	TCustomCheckBox
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Alignment

Controls the position of the check box's caption.

property Alignment: TLeftRight;

Description If *Alignment* is set to *taRightJustify*, the caption appears to the right of the check box. If *Alignment* is set to *taLeftJustify*, the caption appears to the left of the check box.

AllowGrayed

Determines whether check box can be in a "grayed" state.

property AllowGrayed: Boolean;

Description If *AllowGrayed* is set to *True*, the check box has three possible states: checked, unchecked, and grayed. If *AllowGrayed* is set to *False*, the check box has only two possible states: checked and unchecked.

See also *Checked* property, *State* property

Checked

Indicates whether the check box is selected.

property Checked: Boolean;

Description When *Checked* is *True*, a check mark appears in the check box (*State* = *cbChecked*), indicating that the option is selected. When *Checked* is *False*, the check box is either unchecked (*State* = *cbUnchecked*) or grayed (*State* = *cbGrayed*).

See also *AllowGrayed* property, *State* property

State

Indicates whether the check box is selected, deselected, or grayed.

type TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);

property State: TCheckBoxState;

Description These are the possible values of *State*:

Value	Meaning
cbUnchecked	The check box has no check mark, indicating that the user has not selected the option.
cbChecked	The check box has a check mark in it, indicating that the user has selected the option.
cbGrayed	The check box has a check mark in it, but it is grayed.

See also *AllowGrayed* property, *Checked* property

Methods

The following table lists the methods of *TCustomCheckBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Click	protected	TCustomCheckBox
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomCheckBox
CreateParams	protected	TCustomCheckBox
CreateParented	public	TWinControl
CreateWindowHandle	protected	TCustomCheckBox
CreateWnd	protected	TCustomCheckBox
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject

Method	Visibility	Documented in
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Toggle	protected	TCustomCheckBox
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Creates and initializes a *TCustomCheckBox* instance.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to generate a check box at runtime. For check boxes placed on forms at design time, *Create* is called automatically.

CreateParams

Initializes a window-creation parameter record.

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description *CreateParams* calls the inherited method, then makes specific adjustments for check boxes.

See also *TWinControl.CreateParams* method

CreateWindowHandle

Creates a window handle for the check box.

```
procedure CreateWindowHandle(const Params: TCreateParams); override;
```

Description *CreateWindowHandle* implements special processing required by some Windows NT systems, then calls the inherited method.

See also *TWinControl.CreateWindowHandle* method

CreateWnd

Creates a Windows control corresponding to the check box.

```
procedure CreateWnd; override;
```

Description *CreateWnd* calls the inherited method, then sends a BM_SETCHECK message to set the check state of the control.

See also *TWinControl.CreateWnd* method

Toggle

Changes the state of the check box.

```
procedure Toggle; virtual;
```

Description Use the *Toggle* method to select or deselect the check box programmatically. *Toggle* switches among the valid check-box states—from checked to unchecked to grayed, or from checked to unchecked.

See also *AllowGrayed* property, *State* property

TCustomComboBox

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomComboBox

TCustomComboBox is the base class for combo-box components.

Description Use *TCustomComboBox* as a base class when defining a control that represents a specialized Windows combo box. A combo box is an edit box with a scrollable list attached to it; users can select an item from the list or type directly into the edit box. Combo boxes can present the list as a drop-down list or as a list box that is always visible.

Do not create instances of *TCustomComboBox*. To put a combo box in a form, use a descendant of *TCustomComboBox*, such as *TComboBox*.

See also *TComboBox* object, *TDBComboBox* object, *TDriveComboBox* object, *TEdit* object, *TFilterComboBox* object, *TListBox* object

Properties

The following table lists the properties of *TCustomComboBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomComboBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
DropDownCount	protected	8	TCustomComboBox

Property	Visibility	Default	Documented in
DroppedDown	public		TCustomComboBox
EditHandle	protected, read-only		TCustomComboBox
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ItemHeight	protected		TCustomComboBox
ItemIndex	public		TCustomComboBox
Items	public		TCustomComboBox
Left	published		TControl
ListHandle	protected, read-only		TCustomComboBox
MaxLength	protected	0	TCustomComboBox
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	False	TControl
ParentWindow	public		TWinControl
SelLength	public		TCustomComboBox
SelStart	public		TCustomComboBox
SelText	public		TCustomComboBox
ShowHint	public		TControl
Showing	public, read-only		TWinControl
Sorted	protected	False	TCustomComboBox
Style	protected	csDropDown	TCustomComboBox
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Canvas

Canvas provides access to the drawing surface for owner-draw combo-boxes.

property Canvas: TCanvas;

Description Use *Canvas* as the drawing surface when implementing a handler for the *OnDrawItem* event.

See also *OnDrawItem* event, *OnMeasureItem* event, *TCanvas* object

DropDownCount

DropDownCount is the maximum number of items displayed in the drop-down list.

property DropDownCount: Integer;

Description By default, the drop-down list is long enough to contain eight items without requiring the user to scroll to see them all. To make the drop-down list smaller or larger, specify a number larger or smaller than eight as the *DropDownCount* value.

If the *DropDownCount* value is larger than the number of items in the *Items* property, the drop-down list will be just large enough to hold all the possible choices and no larger. If the *DropDownCount* value is smaller then the number of item in the *Items* property, the drop down list will display a scroll bar.

Note *DropDownCount* is only meaningful if the value of *Style* is not *csSimple*.

See also *Items* property, *Style* property

DroppedDown

DroppedDown indicates whether the drop-down list is currently displayed.

property DroppedDown: Boolean;

Description Read *DroppedDown* to determine whether the drop-down list is visible. Set *DropDown* to drop the list down or close it up. *DroppedDown* is *True* when the drop-down list is open, and *False* when it is closed.

Note *DroppedDown* is only meaningful if the value of *Style* is not *csSimple*.

See also *OnDropDown* event, *Style* property

C

EditHandle

EditHandle is the window handle for the edit portion of the combo box.

property EditHandle: HWnd;

Description Read *EditHandle* for Windows API calls that need a handle for the window used by the edit portion of the combo box.

See also *ListHandle* property

ItemHeight

ItemHeight is the height, in pixels, of the items in the drop-down list.

property ItemHeight: Integer;

Description Use *ItemHeight* to specify the height needed to draw the items in the list when *Style* is set to *csOwnerDrawFixed*. If *Style* is *csOwnerDrawVariable*, *ItemHeight* is the default height for drawing list items. When *Style* is *csOwnerDrawVariable*, *ItemHeight* can be overridden by an *OnMeasureItem* event handler. If *Style* is set to any other value, the *ItemHeight* property is inoperative.

See also *OnDrawItem* event, *OnMeasureItem* event, *Style* property

ItemIndex

ItemIndex indicates which item in the drop-down list is selected.

```
property ItemIndex: Integer;
```

Description Read *ItemIndex* to determine which item in the list is the value of the combo box. This value appears in the edit box portion of the combo box. The first item in the list has index 0, the second item has index 1, and so on. If no item is selected, the value of *ItemIndex* is –1.

Set *ItemIndex* to programmatically select an item from the list.

See also *Items* property

Items

Items is a list of the items (strings) in the list portion of the combo box.

```
property Items: TStrings;
```

Description Use *Items* to access the list of items that appear in the combo box list. Use the properties and methods of the *TStrings* object that is the value of *Items* to change the contents of the list or the objects associated with items in the list.

Note Setting the *Items* property assigns the values from a *TStrings* object to the *Items* object. It does not replace the *TStrings* object that is the current value of *Items*.

See also *ItemIndex* property, *TStrings* object, *TStrings.Assign* method

ListHandle

ListHandle is the window handle for the list portion of the combo box.

```
property ListHandle: HWnd;
```

Description Read *ListHandle* for Windows API calls that need a handle for the window used by the list portion of the combo box.

See also *EditHandle* property

MaxLength

MaxLength is the maximum number of characters the user can type into the edit portion of the combo box.

```
property MaxLength: Integer;
```

Description Use *MaxLength* to limit the length of strings the user can type into the combo box. *MaxLength* does not affect the length of strings available in the list. If *MaxLength* is 0, there is no limit to the number of characters that can be entered.

See also *EditHandle* property

SelLength

SelLength is the length, in characters, of the selected text in the edit portion of the combo box.

```
property SelLength: Integer;
```

Description The selected portion of the text in the edit region of the combo box is the string of *SelLength* characters that starts at the character specified by the *SelStart* property. The selected text appears highlighted in the edit region of the combo box.

To select a section of text in the edit region, first set the *SelStart* property to position the cursor, and then set *SelLength* to the number of characters that should be selected. The *SelText* property is the substring that is selected.

Note If the combo box is not the active control, the selected text does not appear highlighted.

See also *SelStart* property, *SelText* property, *TControl.Text* property

C

SelStart

SelStart is the position of the first selected character in the edit box.

```
property SelStart: Integer;
```

Description Read *SelStart* to determine where the selection starts in the *Text* property. The first character is designated with 0, the second with 1, and so on. When *SelLength* is 0, *SelStart* is the position of the cursor.

Set *SelStart* to position the cursor in the edit region of the combo box. Setting *SelStart* to 0 positions the cursor before the first character, setting *SelStart* to 1 positions the cursor before the second character, and so on.

Note Setting *SelStart* changes the *SelLength* property to 0. When using *SelStart* and *SelLength* to programmatically select a portion of the text in the edit region, set *SelStart* before setting *SelLength*.

See also *SelLength* property, *SelText* property, *TControl.Text* property

SelText

SelText is the selected segment of the text in the edit region.

property SelText: **string**;

Description Read *SelText* to obtain the text that is selected in the edit region of the combo box. When the combo box is the active control, *SelText* appears highlighted. Set *SelText* to replace the current selection with a different substring. If no text is currently selected when a value is assigned to *SelText*, the *SelText* string is inserted in the text at the cursor.

See also *SelLength* property, *SelStart* property

Sorted

Sorted determines whether the list portion of the combo box is alphabetized.

property Sorted: Boolean;

Description Set *Sorted* to *True* to sort the items in the list in alphabetical order. When *Sorted* is *True*, new items added to the *Items* list are inserted in the correct sort-order position. When *Sorted* is set to *True*, the original order of list items is lost. Setting *Sorted* back to *False* does not restore the original order, it simply stops the combo box from automatically enforcing the sort order.

See also *Items* property

Style

Style determines the display style of the combo box.

type TComboBoxStyle = (csDropDown, csSimple, csDropDownList, csOwnerDrawFixed, csOwnerDrawVariable);

property Style: TComboBoxStyle;

Description Set *Style* to specify how the list and edit region of the combo box should appear. These are the possible values of the *Style*:

Value	Meaning
csDropDown	Creates a drop-down list with an edit box for manually entered text. All items are strings of the same height.
csSimple	Creates an edit box with a fixed list (list box) underneath. The length of the list is determined by the *Height* of the combo box.
csDropDownList	Creates a drop-down list with no edit box; the user cannot enter text manually. All items are strings of the same height.
csOwnerDrawFixed	Creates an owner-draw drop-down list with an edit box for manually entered text. Each item in the list is the height specified by the *ItemHeight* property.
csOwnerDrawVariable	Creates an owner-draw drop-down list with an edit box for manually entered text. List items can have varying heights.

When *Style* is *csOwnerDrawFixed* or *csOwnerDrawVariable*, the *OnDrawItem* event occurs when the list items need to be drawn. If *Style* is *csOwnerDrawVariable*, an *OnMeasureItem* event occurs as well.

See also *Height* property, *ItemHeight* property, *OnDrawItem* event, *OnMeasureItem* event

Events

The following table lists the events of *TCustomComboBox*:

Event	Visibility	Documented in
OnChange	protected	TCustomComboBox
OnDrawItem	protected	TCustomComboBox
OnDropDown	protected	TCustomComboBox
OnMeasureItem	protected	TCustomComboBox

OnChange

OnChange occurs when the text displayed in the edit region changes.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnChange: TNotifyEvent;
```

Description Write an *OnChange* event handler to take specific action immediately after the user edits the text in the edit region or selects an item from the list. The *Text* property gives the new value in the edit region.

See also *TControl.Text* property

C

OnDrawItem

OnDrawItem occurs when an item in an owner-draw combo box needs to be displayed.

```
type
  TOwnerDrawState = set of (odSelected, odGrayed, odDisabled, odChecked odFocused);
  TDrawItemEvent = procedure(Control: TWinControl; Index: Integer Rect: TRect; State:
    TOwnerDrawState) of object;

property OnDrawItem: TDrawItemEvent;
```

Description Write an *OnDrawItem* event handler to draw the items in the drop-down list of an owner-draw combo box. An *OnDrawItem* event handler can add graphic elements to the list items, or replace the list item text by graphics. *OnDrawItem* passes four parameters to its event handler:

- Control: A reference to the combo box containing the item.
- Index: The index of the item in the *Items* property.
- Rect: A rectangle in which to draw.

- State: The state of the item, which can take one of the following values:

Value	Meaning
odSelected	The item is selected.
odGrayed	The item is grayed.
odDisabled	The item is disabled.
odChecked	The item is checked.
odFocused	The item currently has input focus.

Draw the items on the *Canvas* using the coordinates provided by the *Rect* parameter. *OnDrawItem* occurs only if *Style* is set to *csOwnerDrawFixed* or *csOwnerDrawVariable*.

If an *OnDrawItem* event handler is not provided, the combo box fills the *Rect* parameter with the current brush and writes the text value of the item specified by the *Index* parameter.

See also *Canvas* property, *Items* property, *OnMeasureItem* event, *Style* property

OnDropDown

OnDropDown occurs when the user opens the drop-down list by clicking the arrow at the right of the control.

```
property OnDropDown: TNotifyEvent;
```

Description Write an *OnDropDown* event handler to implement special processing that needs to occur only when the drop-down list is activated.

See also *DroppedDown* property

OnMeasureItem

OnMeasureItem occurs when an item in a *csOwnerDrawVariable* combo box needs to be redisplayed.

```
type TMeasureItemEvent = procedure(Control: TWinControl; Index: Integer; var Height: Integer)
  of object;

property OnMeasureItem: TMeasureItemEvent;
```

Description When *Style* is set to *csOwnerDrawVariable*, the *OnMeasureItem* event precedes *OnDrawItem*. Write an *OnMeasureItem* event handler to specify the height, in pixels, needed to draw an item in the drop-down list.

OnMeasureItem passes three parameters to its event handler:

- Control: A reference to the combo box containing the item.
- Index: The index of the item in the *Items* property.
- Height: The default height of the item, in pixels.

The event handler can reset the final parameter, *Height*, to the appropriate value.

Description *ItemHeight* property, *Items* property, *OnDrawItem* event, *Style* property

Methods

The following table lists the methods of *TCustomComboBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
Change	protected	TCustomComboBox
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomComboBox
ClientToScreen	public	TControl
ComboWndProc	protected	TCustomComboBox
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomComboBox
CreateParams	protected	TCustomComboBox
CreateParented	public	TWinControl
CreateWnd	protected	TCustomComboBox
DefaultHandler	public	TObject
Destroy	public	TCustomComboBox
DestroyComponents	public	TComponent
Destroying	public	TComponent
DestroyWnd	protected	TCustomComboBox
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
DrawItem	protected	TCustomComboBox
DropDown	protected	TCustomComboBox
EnableAlign	public	TWinControl
EndDrag	public	TControl

Method	Visibility	Documented in
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MeasureItem	protected	TCustomComboBox
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectAll	public	TCustomComboBox
SendToBack	public	TControl

Method	Visibility	Documented in
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetStyle	protected	TCustomComboBox
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl
WndProc	protected	TCustomComboBox

Change

Change generates the *OnChange* event.

```
procedure Change; dynamic;
```

Description *Change* is the protected implementation method for the *OnChange* event. Override this method to add additional processing when the text changes or to block the *OnChange* event.

Note When overriding the *Change* method, either call the inherited *Change* method, or the inherited *Changed* method.

See also *OnChange* event, *TControl.Changed* method

Clear

Clear deletes all text from the edit box and all items from the *Items* list.

```
procedure Clear;
```

Description Call *Clear* to empty the combo box at runtime.

See also *Items* property, *TControl.Text* property

ComboWndProc

ComboWndProc provides message responses for the combo box.

```
procedure ComboWndProc(var Message: TMessage; ComboWnd: HWnd; ComboProc: Pointer);
  virtual;
```

Description *ComboWndProc* is the first procedure to receive Windows messages that are sent to the window of the edit region or list portion of the combo box. The *Message* parameter describes the windows message. The *ComboWnd* parameter is the windows handle for the edit region or list. The *ComboProc* parameter is a pointer to the default window procedure for the window that received the message. *ComboWndProc* provides default responses to focus events, mouse events, and key strokes.

Override *ComboWndProc* to change the way the combo box responds to Windows messages. When overriding *ComboWndProc*, be sure to call the inherited method to process all unhandled messages.

See also *EditHandle* property, *ListHandle* property, *TMessage* type, *WndProc* method

Create

Create creates and initializes a *TCustomComboBox* instance.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to generate a combo box at runtime. Combo boxes placed on forms at design time are created automatically.

After calling the inherited constructor, *Create* initializes the following properties:

- *ControlStyle* to [*csCaptureMouse, csSetCaption, csDoubleClicks, csFixedHeight*].
- *DropDownCount* to 8 and *ItemHeight* to 16.
- *Height* to 25 and *Width* to 145.
- *ParentColor* to *False*.
- *Style* to *csDropDown*.
- *TabStop* to *True*.

See also *ControlStyle* property, *DropDownCount* property, *Height* property, *ItemHeight* property, *Style* property, *TabStop* property, *TControl.ParentColor* property, *TWinControl.Create* method, *Width* property

CreateParams

CreateParams initializes a window-creation parameter record.

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description The *CreateWnd* method calls *CreateParams* to initialize the parameters it uses to create the window for the combo box. *CreateParams* overrides the inherited method to implement the *Style* property.

See also *CreateWnd* method, *Style* property

CreateWnd

CreateWnd creates a Windows control corresponding to the combo box.

```
procedure CreateWnd; override;
```

Description *CreateWnd* first calls the inherited *CreateWnd* method, then initializes parameters specific to the combo box control.

See also *CreateParams* method, *TWinControl.CreateWnd* method

Destroy

Destroy destroys the combo box and frees its memory.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Call *Free* instead. *Free* verifies that the combo box is not already freed, and only then calls *Destroy*.

See also *Free* method

DestroyWnd

DestroyWnd destroys the Windows control that represents the combo box.

```
procedure DestroyWnd; override;
```

Description *DestroyWnd* saves the contents of Items list before calling the inherited *DestroyWnd* method. The saved list is used if the window is recreated.

See also *TWinControl.DestroyWnd* method

DrawItem

DrawItem generates the *OnDrawItem* event.

```
procedure DrawItem(Index: Integer; Rect: TRect; State: TOwnerDrawState); virtual;
```

Description When *Style* is *csOwnerDrawFixed* or *csOwnerDrawVariable*, *DrawItem* is called whenever an item in the list needs to be drawn. If an *OnDrawItem* event handler was assigned, *DrawItem* generates an *OnDrawItem* event, passing the *Index*, *Rect*, and *State* parameters to the event handler. Otherwise, *OnDrawItem* fills the rectangle passed in *Rect* and draws any text associated with the indexed item, ignoring the *State* parameter.

Override *DrawItem* to provide class-specific drawing code for the items in the list.

See also *OnDrawItem* event, *Style* property

DropDown

DropDown generates the *OnDropDown* event.

```
procedure DropDown; dynamic;
```

Description *DropDown* is the protected implementation method for the *OnDropDown* event. To perform additional processing or block the *OnDropDown* event, override *DropDown*.

See also *OnDropDown* event

MeasureItem

MeasureItem generates the *OnMeasureItem* event.

```
procedure MeasureItem(Index: Integer; var Height: Integer); virtual;
```

Description *MeasureItem* is the protected implementation method for the *OnMeasureItem* event. Override *MeasureItem* to provide class-specific code to specify the size of list items when *Style* is *csOwnerDrawVariable*.

See also *OnMeasureItem* event, *Style* property

SelectAll

SelectAll selects the entire block of text in the edit box.

```
procedure SelectAll;
```

Description *SelectAll* selects all the text in the edit box. To select only part of the text, use the *SelStart* and *SelLength* properties.

See also *SelLength* property, *SelStart* property, *SelText* property

SetStyle

SetStyle sets the *Style* property.

```
procedure SetStyle(Value: TComboBoxStyle); virtual;
```

Description *SetStyle* is the protected implementation method for the *Style* property. Override *SetStyle* to change the implementation of the *Style* property.

See also *Style* property

WndProc

WndProc processes messages received by the combo box.

```
procedure WndProc(var Message: TMessage); override;
```

Description *WndProc* is the first method to received Windows messages that are sent to the combo box. Messages sent to the child windows of the combo box (for the edit region and the list portion) go to the *ComboWndProc* method instead. *WndProc* handles messages about changes to the system colors defined in the Windows control panel, and closes up the drop-down list when the *Enter* key or the *Escape* key is pressed. All other messages are passed to *Dispatch* for default processing.

See also *ComboWndProc* method, *Dispatch* method, *TWinControl.WndProc* method

TCustomContentProducer

Unit: HTTPApp

TObject • TPersistent • TComponent • TCustomContentProducer

TCustomContentProducer is the base class for all objects that produce the string that is sent as the contents of an HTTP response message.

Description Use *TCustomContentProducer* as a base class when defining a class to produce the contents of an HTTP response message. In most cases, the content producer will assemble a string of HTML commands. However, descendants of *TCustomContentProducer* can be defined to produce any valid MIME type.

See also *TDSTableProducer* object, *TPageProducer* object, *TWebResponse* object

Properties

The following table list the properties of *TCustomContentProducer*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Dispatcher	published		TCustomContentProducer
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
VCLComObject	public		TComponent

Dispatcher

Dispatcher is the dispatcher for the Web server application that includes the *TCustomContentProducer* object.

```
property Dispatcher: TCustomWebDispatcher;
```

Description Use *Dispatcher* to get access to the dispatcher. The properties of the dispatcher give access to the *TWebRequest* and *TWebResponse* objects for the HTTP message that is currently being serviced. Use these objects to obtain detailed information about the message for which the content producer is generating a response string.

See also *TCustomWebDispatcher* object, *TWebRequest* object, *TWebResponse* object

Methods

The following table lists the methods of *TCustomContentProducer*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Content	public	TCustomContentProducer
ContentFromStream	public	TCustomContentProducer
ContentFromString	public	TCustomContentProducer
Create	public	TComponent
DefaultHandler	public	TObject
Destroy	public	TComponent
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TCustomContentProducer
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Content

Content returns the content produced by the content producer.

```
function Content: string; virtual;
```

Description Call *Content* to retrieve the content for using in an HTTP response message. *TCustomContentProducer* returns an empty string. Descendants of *TCustomContentProducer* return the value of the content that they produce. In most cases, this content is a string of HTML commands, although it can be any valid MIME type.

See also *ContentFromStream* method, *ContentFromString* method, *TWebResponse.Content* property

ContentFromStream

ContentFromStream returns the content produced by the content producer in response to input read from a stream.

```
function ContentFromStream(Stream: TStream): string; virtual;
```

Description Call *ContentFromStream* to retrieve the content produced by the content producer based on the information provided by the *Stream* parameter. *TCustomContentProducer* ignores the *Stream* parameter and returns the value created by the *Content* method.

Descendants of *TCustomContentProducer* override this method to interpret streamed input in their own way.

See also *Content* method, *ContentFromString* method, *TWebResponse.Content* property

ContentFromString

ContentFromString returns the content produced by the content producer in response to an input string.

```
function ContentFromString(const S: string): string; virtual;
```

Description Call *ContentFromString* to retrieve the content produced by the content producer based on the information provided by the *S* parameter. *TCustomContentProducer* ignores the string and returns the value created by the *Content* method.

Descendants of *TCustomContentProducer* override this method to interpret the input string in their own way.

See also *Content* method, *ContentFromStream* method, *TWebResponse.Content* property

Notification

Notification overrides the inherited method to respond to messages that the dispatcher is going away.

procedure Notification(AComponent: TComponent; Operation: TOperation); **override**;

Description After calling the inherited method, *Notification* checks whether this call was triggered by the removal of the dispatcher. If so, the *Dispatcher* property is set to **nil**.

See also *Dispatcher* property

TCustomControl

Unit: Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl

The *TCustomControl* component is the abstract base class from which to derive windowed controls that require the support of a *Canvas* for drawing complex visual images.

Description Standard windowed controls already "know" how to display themselves, because they are part of Windows. Thus, a button or a check box component descends directly from *TWinControl*. When creating original windowed controls that do not correspond to standard Windows components, derive them from *TCustomControl*, rather than directly from *TWinControl*. *TCustomControl* makes drawing original controls easier by providing a *Canvas* property to manage the drawing surface and a virtual *Paint* method that is called to respond to WM_PAINT messages.

When creating custom controls that don't need to receive input focus, derive them from *TGraphicControl* instead of *TCustomControl*, to avoid the overhead associated with being a windowed control. Painting a windowed control is slower than painting a graphic control.

See also *TCanvas* object, *TGraphicControl* object, *TWinControl* object

Properties

The following table lists the properties of *TCustomControl*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	protected, read-only		TCustomControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

C

Canvas

Canvas is a *TCanvas* object that presents a drawing surface for the control to draw on.

property Canvas: TCanvas;

Description Use the properties of the *TCanvas* object to draw or paint on the surface of the control. *Canvas* encapsulates a Windows device context, providing all the tools and methods needed for drawing and painting.

Canvas is a protected property that is usually redeclared as public in descendants of *TCustomControl*.

See also *Paint* method, *TCanvas* object

Methods

The following table lists the methods of *TCustomControl*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomControl
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCustomControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject

Method	Visibility	Documented in
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Paint	protected	TCustomControl
PaintTo	public	TWinControl
PaintWindow	protected	TCustomControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TCustomControl*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a custom control at runtime. objects placed on forms at design time are created automatically.

Create creates the canvas for the custom control after calling the inherited constructor.

Nearly every kind of control overrides *Create* to initialize its unique properties. When overriding *the Create* method, always call the inherited *Create* method first, then proceed with the component's initialization.

See also *TWinControl.Create* method

Destroy

Destroy destroys an instance of *TCustomControl*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the instance is not already freed, and only then calls *Destroy*.

Use *Destroy* to free any memory or resources allocated in the *Create* method. When declaring a *Destroy* method in a descendent control type, always add the **override** directive to the declaration and call the inherited *Destroy* as the last statement in the redeclared method.

See also *Free* method

Paint

Paint renders the image of a custom control.

procedure Paint; **virtual**;

Description The *Paint* method for *TCustomControl* does nothing other than provide an interface for a method that responds to WM_PAINT messages. *Paint* is called by the *PaintWindow* method, after that method has supplied the *Canvas* with the handle to a device context. When creating a custom control, always override *Paint* to draw the image of the control.

See also *Canvas* property, *PaintWindow* method, *TWinControl.GetDeviceContext* method

PaintWindow

PaintWindow provides a device context to the canvas and renders the image of the control by calling the *Paint* method.

procedure PaintWindow(DC: HDC); **override**;

Description *PaintWindow* is called automatically when a custom control receives a WM_PAINT message. *PaintWindow* provides a Windows device context for the canvas, calls the *Paint* method, and finally removes the device context. Override *PaintWindow* to provide additional responses to the WM_PAINT message. To draw the image of the control, override the *Paint* method instead.

See also *Canvas* property, *Paint* method, *TWinControl.GetDeviceContext* method

TCustomDataStore

Unit: MXStore

TObject • TPersistent • TComponent • TCustomDataStore

TCustomDataStore is the base class for components that analyze and maintain crosstabulated data.

Description Use *TCustomDataStore* as a base class when defining classes that fetch data from appropriately structured datasets and analyze it into a multidimensional array of summary values. Each dimension corresponds to a field in the dataset over which information is summarized. *TCustomDataStore* maintains a cached projection of this array over a set of active dimensions. The cache allows *TCustomDataStore* to quickly provide a projection of any subset of the active dimensions.

TCustomDataStore introduces properties and methods that describe the dimensions of the crosstabulated data array and the summary values that are computed over those dimensions.

Do not create instances of *TCustomDataStore*. To make the crosstabulated data from a *TCustomDataStore* object available to the objects in a form, use a *TDecisionCube* object.

See also *TDecisionCube* object

Properties

The following table lists the properties of *TCustomDataStore*:

Property	Visibility	Default	Documented in
BinData	public, read-only		TCustomDataStore
Capacity	public		TCustomDataStore
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
CurrentSummary	public		TCustomDataStore
DataSet	public		TCustomDataStore
DesignInfo	public		TComponent
DimensionCount	public, read-only		TCustomDataStore
DimensionMap	public		TCustomDataStore
DimensionMapCount	public, read-only		TCustomDataStore
MaxCells	public		TCustomDataStore
MaxDimensions	public		TCustomDataStore
MaxSummaries	public		TCustomDataStore
Name	published		TComponent
Owner	public, read-only		TComponent
ShowProgressDialog	public		TCustomDataStore
SummaryCount	public, read-only		TCustomDataStore
Tag	published	0	TComponent
VCLComObject	public		TComponent

BinData

BinData indicates whether any of the dimensions in the multidimensional array do not include a cell for every value along the dimension.

property BinData: Boolean;

Description Read *BinData* to determine whether any of the active dimensions have been grouped into ranges or been put into a paged (permanently drilled) state. When *BinData* is *True*, at least one dimension has been paged or has been collapsed into ranges of values. When *BinData* is *False*, all dimensions include entries for every value of the dimension field that appears in the dataset.

See also *TCubeDim.IsBinData* method

Capacity

Capacity limits the memory that may be used by the cached array of summary values.

property Capacity: Integer;

Description Set *Capacity* to limit the number of bytes that may be used by the crosstabulated data store for storing the multidimensional array of summary values. If the number of active dimensions and summaries causes the cache to exceed this value, an *ELowCapacityError* is raised.

See also *ELowCapacityError* object, *OnLowCapacity* event, *TDimensionItem.ActiveType* property

CurrentSummary

CurrentSummary is the index of the current summary from the set of active summaries maintained by the crosstabulated data store.

```
property CurrentSummary: Integer;
```

Description Use *CurrentSummary* to determine which of the active summaries is used to provide values to clients of the data store. *CurrentSummary* returns 0 if the first active summary is the current summary. *CurrentSummary* returns *SummaryCount* – 1 if the last active summary is the current summary.

The dataset specified by the *DataSet* property provides all the fields that make up the dimensions and summaries of the dimension map. At any time, a subset of all the dimensions and summaries provided by the dataset is cached by the crosstabulated data store and available to decision source objects. Of all the active (cached) summaries, only one can be provided to decision source objects at a time. That one summary is the current summary.

See also *DataSet* property, *DimensionMap* property, *GetSummaryName* method, *SummaryCount* property

DataSet

DataSet is the dataset that provides the fields which make up the dimensions and summaries maintained by the crosstabulated data store.

```
property DataSet: TDataSet;
```

Description Set *DataSet* to the dataset that contains the crosstabulated data for the data store. Usually, *DataSet* is a *TDecisionQuery* object. *TDecisionQuery* objects are SQL queries that are grouped appropriately to calculate summaries for a crosstabulated data store.

The *DataSet* property can also be set to a *TQuery* object where the SQL has the same form as that of a *TDecisionQuery* or to any other dataset that contains summarized values in the same structure as the result of a *TDecisionQuery* query.

If the *DataSet* property is a *TQuery* or *TDecisionQuery* object, the crosstabulated data store automatically determines which fields are dimensions and which are summaries. If the *DataSet* property is not a query, the fields in the dimension map must be explicitly identified as dimensions or summaries, and, if they represent summaries, the type of summary must be indicated.

Note The summary values for the crosstabulated data store must be calculated by the *TDataSet* object. The data store does not summarize raw data, but analyzes summarized data into dimensions.

See also *DimensionMap* property, *TDataSet* object, *TDecisionQuery* object, *TQuery* object

DimensionCount

DimensionCount is the number of dimensions cached by the crosstabulated data store.

```
property DimensionCount: Integer;
```

Description Read *DimensionCount* to determine the number of active dimensions. The dataset specified by the *DataSet* property provides all the fields that make up the dimensions and summaries of the dimension map. At any time, a subset of all the dimensions and summaries provided by the dataset is cached by the crosstabulated data store and available to decision source objects. *DimensionCount* is the number of active (cached) dimensions that can be provided to decision source objects.

See also *DataSet* property, *DimensionMap* property, *TDecisionSource* object

DimensionMap

DimensionMap is a set of *TCubeDim* objects that describe the dimensions and summaries maintained by the crosstabulated data store.

```
property DimensionMap: TCubeDims;
```

Description Read *DimensionMap* to access the dimensions and summaries of the data store. The individual entries of the dimension map represent the fields of the dataset specified by the *DataSet* property or calculated fields derived from those fields, such as averages. *DimensionMap* provides information about these fields from the dataset that make up the crosstabulated data store, including the field names and the use of the fields within the multidimensional array.

When the data store is not active, use the properties of the *TCubeDim* objects to change the information cached by the data store. Use the *ActiveType* property of individual *TCubeDim* objects to specify whether those dimensions or summaries from the dataset are cached by the crosstabulated data store. Use the *BinType* property of the *TCubeDim* objects to allow cells along temporal dimensions to represent date ranges or to put the dimensions in a paged (permanently drilled) state.

If the dataset is not a query, use the *DimensionType* property of the *TCubeDim* objects to identify which fields are dimensions and which are summaries, and to identify the type of summary represented by the summary fields.

To change the properties of the entries in the dimension map when the data store is active, use the *Refresh* method.

Note At design time, the *DimensionType*, *Name*, and *ActiveType* properties of the *TCubeDim* objects that make up the dimension map can be set using the Decision Cube Editor.

See also *DataSet* property, *Refresh* method, *TCubeDim.BinType* property, *TDimensionItem.ActiveType* property, *TDiemensionItem.DimensionType* property, *TDimensionItem.Name* property, *TCubeDims* object

DimensionMapCount

DimensionMapCount is the total number of fields from the dataset that make up the dimensions and summaries of the dimension map.

```
property DimensionMap: TCubeDims;
```

Description Read *DimensionMapCount* to determine the number of *TCubeDim* objects that can be accessed through the *DimensionMap* property. The value of *DimensionMapCount* is the number of fields in the dataset specified by the *DataSet* property plus the number of derived fields based on those fields, such as fields for average values that are derived from count and summary fields.

The value of *DimensionMapCount* is not the same as *DimensionCount* + *SummaryCount*. *DimensionCount* and *SummaryCount* count the number of dimensions and summaries that are cached by the crosstabulated data cube and available to the decision source. *DimensionMapCount* also counts any inactive dimensions or summaries that are not included in the cache.

See also *DataSet* property, *DimensionCount* property, *DimensionMap* property, *SummaryCount* property, *TCubeDim* object

MaxCells

MaxCells limits the number of cells that can be included in the cache at any one time.

```
property MaxDimensions: Integer;
```

Description Use *MaxCells* to limit the memory usage of the crosstabulated data store. Memory consumption increases geometrically with each dimension that is cached in the multidimensional array and linearly with each summary. *MaxCells* prevents too many dimensions or summaries from being cached at once.

When the *ActiveType* property of a dimension or summary is *diActive*, it is always added to the cache. When the *ActiveType* property of a dimension or summary is *diAsNeeded*, it can be added only if it does not cause the number of cells to exceed *MaxCells*. The data store swaps in as-needed dimensions and summaries when they are used by the decision source, making room, if necessary, by swapping out closed dimensions or summaries that are not current.

Note Be sure to set *MaxCells* to a value big enough to accommodate all active dimensions and summaries plus the number of as-needed dimensions and summaries that can be open at one time.

See also *MaxDimensions* property, *MaxSummaries* property, *TDimensionItem.ActiveType* property

MaxDimensions

MaxDimensions limits the number of dimensions that can be cached by the crosstabulated data store at any one time.

```
property MaxDimensions: Integer;
```

Description Use *MaxDimensions* to limit the memory usage of the data store. Memory consumption increases geometrically with each dimension that is cached in the multidimensional array. *MaxDimensions* prevents too many dimensions to be cached at once.

When the *ActiveType* property of a dimension is *diActive*, the dimension is always added to the cache. When the *ActiveType* property of a dimension is *diAsNeeded*, it can be added only if it does not cause the number of active dimensions to exceed *MaxDimensions*. The data store swaps in as-needed dimensions when they are open in the decision source, making room, if necessary, by swapping out closed dimensions.

Note Be sure to set *MaxDimensions* to a value big enough to accommodate all active dimensions plus the number of as-needed dimensions that can be open at one time.

See also *DimensionCount* property, *MaxCells* property, *MaxSummaries* property, *TDimensionItem.ActiveType* property

MaxSummaries

MaxSummaries limits the number of summaries that can be cached by the crosstabulated data store at any one time.

```
property MaxSummaries: Integer;
```

Description Use *MaxSummaries* to limit the memory usage of the data store. Memory consumption increases linearly with each summary that is cached in the multidimensional array. *MaxSummaries* prevents too many summaries to be cached at once.

When the *ActiveType* property of a summary is *diActive*, the summary is always added to the cache. When the *ActiveType* property is *diAsNeeded*, it can be added only if it does not cause the number of active summaries to exceed *MaxSummaries*. The data store swaps in as-needed summaries when they are chosen as the current summary, making room, if necessary, by swapping out other as-needed summaries.

See also *MaxCells* property, *MaxDimensions* property, *SummaryCount* property, *TDimensionItem.ActiveType* property

ShowProgressDialog

ShowProgressDialog determines whether a progress bar appears when the cache of crosstabulated values is built.

property ShowProgressDialog: Boolean;

Description Set *ShowProgressDialog* to specify whether a progress bar should appear to provide users with a visual indication when the cache is being built. If the crosstabulated data store includes may dimensions, rebuilding the cache can take a noticeable amount of time. The progress bar provides users with feedback about how long this process will take.

See also *DimensionCount* property, *TProgressBar* object

SummaryCount

SummaryCount is the number of summary fields cached by the crosstabulated data store.

property SummaryCount: Integer;

Description Read *SummaryCount* to determine the number of active summaries in the dimension map. The dataset specified by the *DataSet* property provides all the fields that make up the dimensions and summaries of the dimension map. At any time, a subset of all the dimensions and summaries provided by the dataset is cached by the crosstabulated data store and available to decision source objects. *SummaryCount* is the number of active (cached) summaries that can provide data to decision source objects.

See also *DataSet* property, *DimensionMap* property, *TDecisionSource* object

Events

C

The following table lists the events of *TCustomDataStore*:

Event	Visibility	Documented in
AfterClose	public	TCustomDataStore
AfterOpen	public	TCustomDataStore
BeforeClose	public	TCustomDataStore
BeforeOpen	public	TCustomDataStore
OnLowCapacity	public	TCustomDataStore
OnRefresh	public	TCustomDataStore

AfterClose

AfterClose occurs immediately after the crosstabulated data store has been deactivated.

```
type TCubeNotifyEvent = procedure(DataCube: TCustomDataStore) of object;
property AfterClose: TCubeNotifyEvent;
```

Description Write an *AfterClose* event handler to take specific action when the crosstabulated data store has been shut down. For example, an application could provide feedback about the state of the data store, or update the dimension map for the next time the data store is activated.

See also *BeforeClose* event, *DimensionMap* property

AfterOpen

AfterOpen occurs immediately after the crosstabulated data store has been activated.

```
type TCubeNotifyEvent = procedure(DataCube: TCustomDataStore) of object;
property AfterOpen: TCubeNotifyEvent;
```

Description Write an *AfterOpen* event handler to take specific action when the crosstabulated data store is activated. This is the first opportunity to determine how much memory is consumed by the multidimensional cache maintained by the data store.

See also *BeforeOpen* event, *GetMemoryUsage* method

BeforeClose

BeforeClose occurs immediately before the crosstabulated data store is deactivated.

```
type TCubeNotifyEvent = procedure(DataCube: TCustomDataStore) of object;
property BeforeClose: TCubeNotifyEvent;
```

Description Write a *BeforeClose* event handler to take specific action before the data store shuts down. For example, an application can store information about the current state of the cache before it is cleared and all associated memory freed.

See also *AfterClose* event

BeforeOpen

BeforeOpen occurs immediately before the crosstabulated data store is activated.

```
type TCubeNotifyEvent = procedure(DataCube: TCustomDataStore) of object;
property Before: TCubeNotifyEvent;
```

Description Write a *BeforeOpen* event handler to take specific action before the crosstabulated data store is activated. This is an appropriate time to adjust the value of *Capacity*, or set the *ShowProgressDialog* based on the number of active dimensions in the *DimensionMap*.

This is the last opportunity to adjust the entries in the *DimensionMap* before the cache is built. If the data store has never been active before this, the *DimensionMap* property will be **nil** at this point. A *BeforeOpen* event handler can build a dimension map programmatically if it is **nil**. If the dimension map is not **nil**, the properties of the dimension items can be changed, but the order and number of the dimension items themselves should not be changed.

See also *AfterOpen* event, *Capacity* property, *DimensionMap* property, *ShowProgressDialog* property

OnLowCapacity

OnLowCapacity occurs when the multidimensional cache requires more memory than is allowed by the *Capacity* property.

```
type
  TErrorAction = ( eaFail, eaContinue );
  TCapacityErrorEvent = procedure(var Action: TErrorAction) of object;

property OnLowCapacity: TCapacityErrorEvent;
```

Description Write an *OnLowCapacity* event handler to respond to *ELowCapacityError* exceptions. An *OnLowCapacity* event handler can change the value of *Capacity* or deactivate some of the dimensions in the dimension map in order to allow the cache to fit in the available memory. If the memory problem is handled by the *OnLowCapacity* event handler, change the *Action* parameter to *eaContinue* to allow the data store to continue building the cache. Otherwise, an *ELowCapacityError* exception will be raised.

See also *Capacity* property, *DimensionMap* property, *ELowCapacityError* object

C

OnRefresh

OnRefresh occurs immediately before the dimension map is changed.

```
type TCubeRefreshEvent = procedure(DataCube: TCustomDataStore; DimMap: TCubeDims)
  of object;

property OnRefresh: TCubeRefreshEvent;
```

Description Write an *OnRefresh* event handler to take specific action when the dimension map changes. The *DimMap* parameter specifies the new value that is about to be set for the *DimensionMap* property. Use an *OnRefresh* event handler to change the properties of these dimension items before the *DimensionMap* property is set and the cache is rebuilt. This is also an opportunity to reset the *MaxDimensions* or *MaxSummaries* property to accommodate the dimension map that is about to be set.

See also *DimensionMap* property, *MaxDimensions* property, *MaxSummaries* property, *Refresh* method

Methods

The following table lists the methods of *TCustomDataStore*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TCustomDataStore
DefaultHandler	public	TObject
Destroy	public	TCustomDataStore
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetDimensionMemberCount	public	TCustomDataStore
GetDimensionName	public	TCustomDataStore
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetMemberAsString	public	TCustomDataStore
GetMemberAsVariant	public	TCustomDataStore
GetMemoryUsage	public	TCustomDataStore
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetSummaryAsString	public	TCustomDataStore
GetSummaryAsVariant	public	TCustomDataStore
GetSummaryName	public	TCustomDataStore
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject

Method	Visibility	Documented in
MethodName	public	TObject
NewInstance	public	TObject
Refresh	public	TCustomDataStore
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Create

Create creates an instance of *TCustomDataStore*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create an instance of *TCustomDataStore* at runtime. Create allocates the helper objects used by the crosstabulated data store to manage the dimension map and the crosstabulated data.

See also *TComponent.Create* method

Destroy

Destroy destroys an instance of *TCustomDataStore*.

destructor Destroy; **override**;

Description Do not call the *Destroy* method directly. Instead call *Free*. *Free* verifies that the memory for the decision cube is not already freed, and only then calls *Destroy*.

See also *Free* method

C

GetDimensionMemberCount

GetDimensionMemberCount returns the number of distinct values for a dimension of the data store.

function GetDimensionMemberCount(Dimension: Integer): Integer; **virtual**;

Description Call *GetDimensionMemberCount* to determine the number of distinct field values in the dimension specified by the *Dimension* parameter. Zero specifies the first active dimension, and *DimensionCount* – 1 specifies the last active dimension.

To obtain the individual values of the dimension, use the *GetMemberAsString* or *GetMemberAsVariant* method. *GetDimensionMemberCount* returns a value that provides an upper limit to the *Index* parameter of the *GetMemberAsString* or *GetMemberAsVariant* method.

To obtain the name of the dimension referred to by the *Dimension* parameter, use the *GetDimensionName* method.

See also *DimensionCount* property, *GetDimensionName* method, *GetMemberAsString* method, *GetMemberAsVariant* method

GetDimensionName

GetDimensionName returns the name of an active dimension.

```
function GetDimensionName( Dimension: Integer ): String; virtual;
```

Description Call *GetDimensionName* to obtain the name of the dimension indexed by the *Dimension* parameter. Zero specifies the first active dimension and *DimensionCount* – 1 specifies the last active dimension. The name of a dimension is the name of the database field the dimension represents, unless the *Name* property of the corresponding *TDimensionItem* object is changed.

See also *DimensionCount* property, *TDimensionItem.Name* property

GetMemberAsString

GetMemberAsString returns the specified value from a dimension of the data store, formatted as a string.

```
function GetMemberAsString( Dimension, Index: Integer ): String; virtual;
```

Description Call *GetMemberAsString* to retrieve one of the values from the dimension specified by the *Dimension* parameter. Zero specifies the first dimension, and *DimensionCount* –1 specifies the last dimension. The *Index* parameter indicates which value along the dimension is returned, where 0 is the first value. *GetDimensionMemberCount* returns the number of values included in the dimension.

GetMemberAsString returns a field value formatted as a string. To obtain the field value as an unformatted variant, use the *GetMemberAsVariant* method.

See also *DimensionCount* property, *GetDimensionMemberCount* method, *GetDimensionName* method, *GetMemberAsVariant* method

GetMemberAsVariant

GetMemberAsVariant returns the specified value from a dimension of the crosstabulated data store.

```
function GetMemberAsVariant( Dimension, Index: Integer ): Variant;
```

Description Call *GetMemberAsVariant* to retrieve one of the values from the dimension specified by the *Dimension* parameter. Zero specifies the first dimension, and *DimensionCount* –1 specifies the last dimension. The *Index* parameter indicates which value along the dimension is returned, where 0 is the first value. *GetDimensionMemberCount* returns the number of values included in the dimension.

See also *DimensionCount* property, *GetDimensionMemberCount* method, *GetDimensionName* method, *GetMemberAsString* method

GetMemoryUsage

GetMemoryUsage returns the number of bytes of memory used to cache the multidimensional array of summaries along all active dimensions.

```
function GetMemoryUsage: Integer;
```

Description Call *GetMemoryUsage* to determine the size of the cache that stores the crosstabulated information for the data store. If the returned value is too large, consider making some of the active dimensions paged by setting their *BinType* property to *binSet* or allowing the data store to swap some dimensions out by setting their *ActiveType* property to *diAsNeeded*. The cube dimension can be accessed using the *DimensionMap* property.

The storage needs of the crosstabulated data store increase geometrically with each active dimension that is added. Applications must find the best balance between the performance benefit of caching many dimensions so that they are immediately available to decision controls, and the need to keep the number of active dimensions small enough so that the cache does not consume inordinate amounts of memory.

See also *DimensionMap* property, *TCubeDim.BinType* property, *TDimensionItem.ActiveType* property

GetSummaryAsString

GetSummaryAsString returns the value of the current summary for a set of values along all the active dimensions.

```
function GetSummaryAsString( Coord: TSmallIntArray ): String; virtual;
```

Description Call *GetSummaryAsString* to obtain the value of a single cell from the multidimensional array of summary values. Set the *Coord* parameter to an array of indexes, where each index identifies a value along one of the dimensions of the array. The first value of *Coord* is the index of the desired value along the first active dimension, the second value of *Coord* is the index of the desired value along the second active dimension, and so on. *Coord* should contain *DimensionCount* values.

To obtain the names of the active dimensions that correspond to the entries in the *Coord* parameter, use the *GetDimensionName* method. To obtain the name of the specified value along any of the dimensions, use the *GetMemberAsString* method.

GetSummaryAsString formats the summarized value as a string. To obtain the summary as a variant, use the *GetSummaryAsVariant* method.

See also *CurrentSummary* property, *DimensionCount* property, *GetDimensionName* method, *GetMemberAsString* method, *GetSummaryAsVariant* method

C

GetSummaryAsVariant

GetSummaryAsVariant returns the value of the current summary for a set of values along all the active dimensions.

```
function GetSummaryAsVariant( Coord: TSmallIntArray ): Variant; virtual;
```

Description Call *GetSummaryAsVariant* to obtain the value of a single cell from the multidimensional array of summary values. Set the *Coord* parameter to an array of indexes, where each index identifies a value along one of the dimensions of the array. The first value of *Coord* is the index of the desired value along the first active dimension, the second value of *Coord* is the index of the desired value along the second active dimension, and so on. *Coord* should contain *DimensionCount* values.

To obtain the names of the active dimensions that correspond to the entries in the *Coord* parameter, use the *GetDimensionName* method. To obtain the name of the specified value along any of the dimensions, use the *GetMemberAsString* method.

To obtain the summary formatted as a string, use the *GetSummaryAsString* method.

See also *CurrentSummary* property, *DimensionCount* property, *GetDimensionName* method, *GetMemberAsString* method, *GetSummaryAsString* method

GetSummaryName

GetSummaryName returns the name of an active summary.

```
function GetSummaryName( ISum: Integer ): String; virtual;
```

Description Call *GetSummaryName* to obtain the name of the summary specified by the *ISum* parameter. *Zero* specifies the first active summary and *SummaryCount* – 1 specifies the last active summary.

See also *SummaryCount* property

Refresh

Refresh updates the dimension map of the crosstabulated data store.

```
function Refresh( DimensionMap : TCubeDims; bForce: Boolean): Boolean;
```

Description To change the properties of the dimensions in the dimension map, make a copy of the *DimensionMap* property, make any desired changes to the properties of the *TCubeDim* objects, and call *Refresh*, passing in the altered dimension map. The *bForce* parameter specifies whether the cache should be rebuilt unconditionally. When *bForce* is *False*, the cache will not be rebuilt unless the changes require it.

When the data store is active, call *Refresh* rather than directly altering the dimension map. Do not call *Refresh* while the data store is building the cache.

Use the *BeforeOpen* and *AfterOpen* events to determine when the cache is being rebuilt.

See also *AfterOpen* event, *BeforeOpen* event, *DimensionMap* property, *TCubeDim* object, *TCubeDims* object

TCustomDBGrid

Unit: DBGrids

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomGrid • TCustomDBGrid

TCustomDBGrid is the abstract base class for grid controls that display the records from a dataset in a tabular format.

Description *TCustomDBGrid* introduces new properties, events, and methods to expand the capabilities of *TCustomGrid* for displaying the fields from a dataset. At runtime, users can manipulate *TCustomDBGrid* objects using the *TDBNavigator* object.

Do not create instances of *TCustomDBGrid*. Use *TCustomDBGrid* as a base class when declaring grid objects that display information from datasets. Properties and methods of *TCustomDBGrid* provide basic behavior that descendent classes inherit as well as behavior that components can override to customize their behavior.

See also *TDBGrid* object, *TDataSet* object, *TDBNavigator* object

Properties

The following table lists the properties of *TCustomDBGrid*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Columns	protected		TCustomDBGrid
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DataLink	protected, read-only		TCustomDBGrid
DataSource	protected		TCustomDBGrid
DefaultDrawing	protected	True	TCustomDBGrid
DesignInfo	public		TComponent
EditorMode	public		TCustomGrid
Enabled	public	True	TControl
FieldCount	public, read-only		TCustomDBGrid
Fields	public, read-only		TCustomDBGrid
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
IndicatorOffset	protected, read-only		TCustomDBGrid
LayoutLock	protected, read-only		TCustomDBGrid
Left	published		TControl
Name	published		TControl
Options	protected	[dgEditing, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgConfirmDelete, dgCancelOnExit]	TCustomDBGrid
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	False	TControl
ParentWindow	public		TWinControl
ReadOnly	protected	False	TCustomDBGrid
SelectedField	public		TCustomDBGrid
SelectedIndex	public		TCustomDBGrid
SelectedRows	protected, read-only		TCustomDBGrid
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent

Property	Visibility	Default	Documented in
TitleFont	protected		TCustomDBGrid
Top	published		TControl
UpdateLock	protected, read-only		TCustomDBGrid
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Columns

TColumns describes the display attributes and field bindings of the columns in the *TCustomDBGrid* object.

property Columns: TDBGridColumns;

Description Use *Columns* to read or set the field bindings and display attributes of the columns in the data-aware grid. *Columns* is an indexed collection of *TColumn* objects. Use the properties of the *TColumn* objects to specify the display attributes or field bindings of individual columns in the grid. The field binding of a column designates a field within the dataset specified by the *DataSource* property.

Columns can be set at design time through the Columns editor, or programmatically at runtime.

See also *CreateColumns* method, *Fields* property, *IndicatorOffset* property, *SetColumnAttributes* method, *StoreColumns* method, *TDBGridColumns* object, *TColumn* object

C

DataLink

DataLink helps the data-aware grid manage its link to the data source and respond to data events.

property DataLink: TGridDataLink;

Description Applications should not need to use the *DataLink* property. It is exposed as a protected method so that component writers can create data grids that use descendants of *TGridDataLink* to respond to additional data events.

DataLink helps *TCustomDBGrid* to

- Resolve the field bindings specified by *Columns* with the fields in the dataset.
- Respond to changes in the values of fields.
- Respond to changes in record currency.
- Respond to changes in the editing state of the dataset.
- Keep track of unposted edits and discard or post them as appropriate.

See also *Columns* property, *DataSource* property, *Fields* property, *TGridDataLink* object

DataSource

DataSource identifies the link to the dataset where the data-aware grid finds its data.

property DataSource: TDataSource;

Description Set *DataSource* to the *TDataSource* object that links to the dataset where the data-aware grid should fetch its data. *DataSource* allows the data-aware grid to read from, write to, and navigate around the dataset, while maintaining record currency with other data-aware objects that use the same data source.

See also *DataLink* property, *Fields* property, *TDataSource* object

DefaultDrawing

DefaultDrawing indicates whether the cells in the data-aware grid are drawn automatically, or if the application provides customized painting using the *OnDrawColumnCell* event.

property DefaultDrawing: Boolean;

Description Set *DefaultDrawing* to *True* to allow the data-aware grid to draw the data in the cells of the grid automatically. Set *DefaultDrawing* to *False* to turn off the default drawing when providing customized painting in an *OnDrawColumnCell* event handler or an *OnDrawDataCell* event handler.

When *DefaultDrawing* is *True*, the data-aware grid fetches the data for each cell from the appropriate entry in the *Fields* array, and writes it to the cell. It then draws the focus rectangle if the cell has focus, or if the cell is selected and the *Options* property includes *dgAlwaysShowSelection*.

When *DefaultDrawing* is *False*, the data-aware grid draws the appropriate background color on the cell and sets up the brush and font described by the appropriate *TColumn* object. However, the value of the field is not written to the cell, and no focus rectangle is drawn.

See also *Columns* property, *Fields* property, *OnDrawColumnCell* event, *OnDrawDataCell* event, *TColumn* object

FieldCount

FieldCount is the number of fields (columns) displayed in the grid.

property FieldCount: Integer;

Description Use *FieldCount* to iterate through all the field components indexed by the *Fields* property. *FieldCount* may differ from the number of fields in the underlying dataset, because the grid may contain calculated fields, and not all fields in the dataset are necessarily shown in the grid. Individual fields are

removed and calculated fields are added using the Columns editor at design time.

FieldCount is the same as the number of columns at runtime. At design time, the grid may contain empty columns.

See also *Columns* property, *Fields* property

Fields

Fields is an indexed array of field components that correspond to the columns of the data-aware grid.

```
property Fields[Index: Integer]: TField;
```

Description Use *Fields* to gain direct access to the field component in the dataset for a particular column in the grid. The field for the first data column is obtained using an *Index* of 0, the field for the second data column using an *Index* of 1, and so on. The range for *Index* is 0.. *FieldCount* – 1.

The first data column in the grid may not be the same as the first column in the grid. If *Options* includes *dgIndicator*, the first column in the grid will be a non-scrolling columns that contains the current row indicator.

The individual field components in *Fields* can be used to obtain information such as the value of the field on the current record, its preferred display width, or the string to draw when displaying or editing the field. Use the field component to write values directly to the dataset without using the editor in the cell.

Note If the column in the grid represents a calculated field, there is no field component for that column in the dataset. *Fields* returns **nil** when the *Index* specifies the column that contains a calculated field. To obtain the field component for a calculated field, use the *GetColField* method.

See also *Columns* property, *GetColField* method, *Options* property, *SelectedField* property, *TField* object

IndicatorOffset

IndicatorOffset is the index of the first column in the grid that contains data.

```
property IndicatorOffset: Byte;
```

Description Use *IndicatorOffset* to convert between the positions of the columns in the *Columns* property and the columns drawn on the data-aware grid. *IndicatorOffset* is 1 if the *Options* property includes *dgIndicator*, because the first column of the grid is used for the current row indicator. *IndicatorOffset* is 0 if all columns contain data.

See also *Columns* property, *DataToRawColumn* method, *Options* property, *RawToDataColumn* method

LayoutLock

LayoutLock is the number of times the *BeginLayout* method has been called without an *EndLayout* or *CancelLayout* call.

```
property LayoutLock: Byte;
```

Description Use *LayoutLock* to determine when the column bindings change. Whenever a column width or binding changes or the fields in the data source change, the *BeginLayout* method is called. *BeginLayout* increments *LayoutLock* to keep track of the fact that the column bindings are changing. When changes are complete, the *EndLayout* method is called. *EndLayout* decrements *LayoutLock*. When *LayoutLock* returns to 0, the data-aware grid knows that all changes are complete, and the *Fields* and *Columns* properties are updated.

See also *AcquireLayoutLock* method, *BeginLayout* method, *CancelLayout* method, *DeferLayout* method, *DefineFieldMap* method, *EndLayout* method

Options

Options specifies various display and behavioral properties of the data-aware grid.

```
type
  TDBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles, dgIndicator, dgColumnResize,
    dgColLines, dgRowLines, dgTabs, dgRowSelect,dgAlwaysShowSelection, dgConfirmDelete,
    dgCancelOnExit, dgMultiSelect);
  TDBGridOptions = set of TDBGridOption;

property Options: TDBGridOptions;
```

Description Set *Options* to include the desired properties for the data-aware grid. *Options* is a set drawn from the following values:

Value	Meaning
dgEditing	The user can edit data using the grid. *dgEditing* is ignored if *Options* includes *dgRowSelect*.
dgAlwaysShowEditor	The grid is always in edit mode. That is, the user does not have to press *Enter* or *F2* before editing the contents of a cell. *dgAlwaysShowEditor* does nothing unless *dgEditing* is also included in *Options*. *dgAlwaysShowEditor* is ignored if *Options* includes *dgRowSelect*.
dgTitles	Titles appear at the top of the columns in the grid.
dgIndicator	A small pointer appears in the first column to indicate which row is current.
dgColumnResize	Columns that are bound to fields can be resized or moved.
dgColLins	Lines appear between columns in the grid.
dgRowLines	Lines appear between the rows of the grid.
dgTabs	The user can navigate through the grid using the *Tab* and *Shift+Tab* keys.

Value	Meaning
dgRowSelect	The user can select an entire row, as well as selecting individual cells. If *Options* includes *dgRowSelect*, *dgEditing* and *dgAlwaysShowEditor* are ignored.
dgAlwaysShowSelection	The selected cell displays the focus rectangle even when the grid does not have focus.
dgConfirmDelete	A message box appears, asking for confirmation, when the user types *Ctrl+Delete* to delete a row in the grid.
dgCancelOnExit	When the user exits the grid from an inserted record to which the user made no modifications, the inserted record is not posted to the dataset. This prevents the inadvertent posting of empty records.
dgMultiSelect	More than one row in the grid can be selected at a time.

See also *ReadOnly* property, *SelectedIndex* property, *SelectedRows* property

ReadOnly

ReadOnly indicates whether the grid is used for display only, or whether the user can edit data using the grid.

```
property ReadOnly: Boolean;
```

Description Set *ReadOnly* to *True* to prevent users from changing the data in the dataset. Set *ReadOnly* to *False* to allow users to edit data using the grid.

When *ReadOnly* is *True*, users cannot use the *Insert* key to insert a new row in the grid or use the *Down Arrow* key to add a row to the end of the grid.

See also *CanEditModify* method, *Options* property

SelectedField

SelectedField is the field component for the currently selected cell in the grid.

```
property SelectedField: TField;
```

Description Set *SelectedField* to move focus to a particular field in the grid. Read *SelectedField* to obtain access to the field component for the currently selected cell. If there is not currently selected cell, *SelectedField* is **nil**. For example, when an entire row is selected, *SelectedField* is **nil**.

See also *Fields* property, *SelectedIndex* property

SelectedIndex

SelectedIndex is the index of the currently selected column in the *Columns* array.

```
property SelectedIndex: Integer;
```

Description Set *SelectedIndex* to move focus to a column in the grid that is identified by position. Read *SelectedIndex* to determine which column in the grid has focus. A value of 0 indicates the first data column, 1 is the second data column, and so on. *SelectedIndex* is –1 if there is no currently selected column.

If the *Options* property includes *dgIndicator*, the index of the data column given by *SelectedIndex* will differ from the index of the physical column in the grid.

To access the field component for the selected column, use the *SelectedField* property.

See also *Columns* property, *Options* property, *SelectedField* property

SelectedRows

SelectedRows is a set of bookmarks to all the records in the dataset that correspond to rows selected in the grid.

```
property SelectedRows: TBookmarkList;
```

Description Use the properties and methods of the *TBookmarkList* object returned by *SelectedRows* to

- Determine the number of rows in the grid that are selected.
- Determine whether the current record in the dataset is selected.
- Determine whether a particular record in the dataset is selected.
- Delete all selected rows from the dataset.

SelectedRows is only meaningful when the *Options* property includes *dgRowSelect*. *SelectedRows* can only identify a single row unless the *Options* property includes *dgMultiSelect*.

See also *Options* property, *TBookmarkList* object

TitleFont

TitleFont describes the font used to draw the column titles in the grid.

```
property TitleFont: TFont;
```

Description Set *TitleFont* to change the font used to draw the column headings. *TitleFont* is only meaningful if the *Options* property includes *dgTitles*.

See also *Options* property, *TFont* object

UpdateLock

UpdateLock counts the number of times the *BeginUpdate* method is called without a corresponding call to *EndUpdate*.

```
property UpdateLock: Byte;
```

Description *UpdateLock* prevents the grid from repainting while the data in the grid changes. Whenever a change is made that affects the data that is drawn in the grid, the *BeginUpdate* method is called. *BeginUpdate* increments *UpdateLock* to keep track of the fact that the contents of the grid are changing. When changes are complete, the *EndUpdate* method is called. *EndUpdate* decrements *UpdateLock*.

When *UpdateLock* returns to 0, the data-aware grid can repaint its cells. When *UpdateLock* is greater than 0, the grid defers repaints. *UpdateLock* prevents the grid from flickering when the grid is resized or the user scrolls through the data.

See also *BeginUpdate* method, *EndUpdate* method

Events

The following table lists the events of *TCustomDBGrid*:

Event	Visibility	Documented in
OnCellClick	protected	TCustomDBGrid
OnColEnter	protected	TCustomDBGrid
OnColExit	protected	TCustomDBGrid
OnColumnMoved	protected	TCustomDBGrid
OnDrawColumnCell	protected	TCustomDBGrid
OnDrawDataCell	protected	TCustomDBGrid
OnEditButtonClick	protected	TCustomDBGrid
OnTitleClick	protected	TCustomDBGrid

OnCellClick

OnCellClick occurs when the user releases the mouse in one of the cells of the grid.

```
type TDBGridClickEvent = procedure (Column: TColumn) of object;

property OnCellClick: TDBGridClickEvent;
```

Description Write an *OnCellClick* event handler to take specific action when the user clicks in one of the cells of the data-aware grid. The *Column* parameter is the *TColumn* object that corresponds to the column where the mouse was when the user released the left mouse button.

OnCellClick occurs when the user releases the mouse. To take action when the user clicks down on the mouse, use the *OnColEnter* event.

OnCellClick will not occur when the mouse is released at the end of resizing a row or column, or when the cell is being edited.

See also *Columns* property, *OnColEnter* event, *OnTitleClick* event, *TColumn* object

OnColEnter

OnColEnter occurs when focus moves to a new cell in the grid.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnColEnter: TNotifyEvent;
```

Description Write an *OnColEnter* event handler to take specific action when a new cell has just been selected. Focus moves to a cell when

- The user navigates to the cell using the keyboard. For example, when the user uses the *Tab* key, or the *Home* key.
- The user clicks the mouse button down in the cell.
- The *SelectedField* or *SelectedIndex* property is set.

Read the *SelectedField* or *SelectedIndex* property to determine which cell was just entered.

See also *OnCellClick* event, *OnColExit* event, *SelectedField* property, *SelectedIndex* property

OnColExit

OnColExit occurs just before a cell in the grid loses input focus.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnColExit: TNotifyEvent;
```

Description Write an *OnColExit* event handler to take specific action when a cell is about to lose focus. For example, use *OnColExit* to post edits to the underlying dataset when the user leaves a cell, rather than waiting for the user to leave the record.

A cell loses focus when

- The user navigates out of the cell using the keyboard. For example, when the user presses the *Tab* key or the *Home* key.
- The user clicks the mouse button down in another cell.
- The *SelectedField* or *SelectedIndex* property is set.

When the selected cell in the data-aware grid changes, first the *OnColExit* event occurs. Then the values of *SelectedField* and *SelectedIndex* are updated. Finally the *OnColEnter* event occurs. Thus, in an *OnColExit* event handler, the *SelectedField* or *SelectedIndex* property can be used to determine which column is about to lose focus.

See also *OnColEnter* event, *SelectedField* property, *SelectedIndex* property

OnColumnMoved

OnColumnMoved occurs when the user moves a column using the mouse.

```
type TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: Longint) of object;

property OnColumnMoved: TMovedEvent;
```

Description Write an *OnColumnMoved* event handler to take specific action just after a column in the grid has been moved. The *FromIndex* parameter gives the position the column previously occupied in the *Columns* array. The *ToIndex* parameter gives the position the column now occupies.

OnColumnMoved can only occur when the *Options* property includes *dgColumnResize*.

See also *Columns* property, *Options* property

OnDrawColumnCell

OnDrawColumnCell occurs when the grid needs to paint a cell.

```
type
  TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
  TDrawColumnCellEvent = procedure (Sender: TObject; const Rect: TRect;
   DataCol: Integer; Column: TColumn; State: TGridDrawState) of object;

property OnDrawColumnCell: TDrawColumnCellEvent;
```

Description Write an *OnDrawColumnCell* event handler to provide customized drawing for the data in the cells of the grid. Draw on the cell using the methods of the *Canvas* property. The *Rect* parameter indicates the location of the cell on the canvas. The *DataCol* parameter is the index of the column in the *Columns* array. The *Column* parameter is the *TColumn* object that describes the display attributes and field binding for the cell. The *State* parameter indicates whether the cell has input focus, whether the cell is selected, and whether the cell is a fixed (non-data) cell such as a column header.

An *OnDrawColumnCell* event handler can call the *DefaultDrawColumnCell* method to instruct the data-aware grid to write the data value in the cell. The *OnDrawColumnCell* event handler can augment the *DefaultDrawColumnCell* method to provide visual indications of selection and focus. If the *OnDrawColumnCell* event handler does not use *DefaultDrawColumnCell* to draw the text for the cell, use the *GetFieldValue* method to get the text that should be displayed in the cell.

If *DefaultDrawing* is *True*, the data will already be drawn in the cell before the *OnDrawColumnCell* event, and the grid draws a focus rectangle around selected cells after the *OnDrawColumnCell* event.

Note If the *Columns* property has a *State* property of *csDefault*, the *OnDrawDataCell* event occurs before the *OnDrawColumnCell*. The *OnDrawDataCell* event is obsolete, and only included for backward compatibility. Do not set both an *OnDrawDataCell* event handler and an *OnDrawColumnCell* event handler.

See also *DefaultDrawColumnCell* method, *DefaultDrawing* property, *DrawCell* method, *GetFieldValue* method, *OnDrawDataCell* event, *TCustomControl.Canvas* property, *TDBGridColumns.State* property

OnDrawDataCell

OnDrawDataCell occurs when the grid needs to paint a cell if the *State* property of *Columns* is *csDefault*.

```
type
  TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
  TDrawDataCellEvent = procedure (Sender: TObject; const Rect: TRect; Field: TField;
   State: TGridDrawState) of object;

property OnDrawDataCell: TDrawDataCellEvent;
```

Description Do not write an *OnDrawDataCell* event handler. *OnDrawDataCell* is obsolete and included for backward compatibility. Instead, write an *OnDrawColumnCell* event handler.

See also *DefaultDrawDataCell* method, *DefaultDrawing* property, *OnDrawColumnCell* event, *TDBGridColumns.State* property

OnEditButtonClick

OnEditButtonClick occurs when the user presses the ellipsis button in a grid column.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnEditButtonClick: TNotifyEvent;
```

Description Write an *OnEditButtonClick* event handler to bring up an appropriate list or dialog when the user presses the ellipsis button in a grid cell. The *OnEditButtonClick* event handler can set the value of the field based on the user response to the list or dialog. Use the *SelectedField* property to access the *TField* component whose value should be set.

See also *OnCellClick* event, *SelectedField* property, *TField* object

OnTitleClick

OnTitleClick occurs when the user releases the mouse in one of the column headers.

```
type TDBGridClickEvent = procedure (Column: TColumn) of object;

property OnTitleClick: TDBGridClickEvent;
```

Description Write an *OnTitleClick* event handler to take specific action when the user clicks in one of the column headers. The *Column* parameter is the *TColumn* object that corresponds to the column where the mouse was when the user released the left mouse button.

See also *Columns* property, *OnCellClick* event, *TColumn* object

Methods

The following table lists the methods of *TCustomDBGrid*:

Method	Visibility	Documented in
AcquireLayoutLock	protected	TCustomDBGrid
Assign	public	TPersistent
BeginDrag	public	TControl
BeginLayout	protected	TCustomDBGrid
BeginUpdate	protected	TCustomDBGrid
BringToFront	public	TControl
Broadcast	public	TWinControl
CancelLayout	protected	TCustomDBGrid
CanEditAcceptKey	protected	TCustomDBGrid
CanEditModify	protected	TCustomDBGrid
CanEditShow	protected	TCustomDBGrid
CanFocus	public	TWinControl
CellClick	protected	TCustomDBGrid
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ColEnter	protected	TCustomDBGrid
ColExit	protected	TCustomDBGrid
ColumnMoved	protected	TCustomDBGrid
ColWidthsChanged	protected	TCustomDBGrid
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomDBGrid
CreateColumns	protected	TCustomDBGrid
CreateEditor	protected	TCustomDBGrid
CreateParented	public	TWinControl
CreateWnd	protected	TCustomDBGrid
DataToRawColumn	protected	TCustomDBGrid
DefaultDrawColumnCell	public	TCustomDBGrid
DefaultDrawDataCell	public	TCustomDBGrid
DefaultHandler	protected	TCustomDBGrid
DeferLayout	protected	TCustomDBGrid

Method	Visibility	Documented in
DefineFieldMap	protected	TCustomDBGrid
Destroy	public	TCustomDBGrid
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
DrawCell	protected	TCustomDBGrid
DrawColumnCell	protected	TCustomDBGrid
DrawDataCell	protected	TCustomDBGrid
EditButtonClick	protected	TCustomDBGrid
EnableAlign	public	TWinControl
EndDrag	public	TControl
EndLayout	protected	TCustomDBGrid
EndUpdate	protected	TCustomDBGrid
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetColField	protected	TCustomDBGrid
GetEditLimit	protected	TCustomDBGrid
GetEditMask	protected	TCustomDBGrid
GetEditText	protected	TCustomDBGrid
GetFieldValue	protected	TCustomDBGrid
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
HighlightCell	protected	TCustomDBGrid
InheritsFrom	public	TObject

Method	Visibility	Documented in
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
KeyDown	protected	TCustomDBGrid
KeyPress	protected	TCustomDBGrid
LayoutChanged	protected	TCustomDBGrid
LinkActive	protected	TCustomDBGrid
Loaded	protected	TCustomDBGrid
MethodAddress	public	TObject
MethodName	public	TObject
MouseCoord	public	TCustomGrid
MouseDown	protected	TCustomDBGrid
MouseUp	protected	TCustomDBGrid
NewInstance	public	TObject
Notification	protected	TCustomDBGrid
PaintTo	public	TWinControl
Perform	public	TControl
RawToDataColumn	protected	TCustomDBGrid
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
Scroll	protected	TCustomDBGrid
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetColumnAttributes	protected	TCustomDBGrid
SetEditText	protected	TCustomDBGrid
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
StoreColumns	protected	TCustomDBGrid
TimedScroll	protected	TCustomDBGrid
TitleClick	protected	TCustomDBGrid
Update	public	TWinControl

Method	Visibility	Documented in
UpdateControlState	public	TWinControl
ValidFieldIndex	public	TCustomDBGrid

AcquireLayoutLock

AcquireLayoutLock calls *BeginLayout* if the table is not currently being reformatted.

```
function AcquireLayoutLock: Boolean;
```

Description Applications cannot call this protected method. *AcquireLayoutLock* is called before recomputing *Columns* when the fields in the dataset change or when the *Columns* property changes. *AcquireLayoutLock* returns *True* if the table is not already being laid out, to indicate that the computation of *Columns* should proceed. *AcquireLayoutLock* prevents the computation of *Columns* from looping endlessly because of the changes that must occur during the computation.

See also *BeginLayout* method, *Columns* property, *LayoutChanged* property

BeginLayout

BeginLayout increments the *LayoutLock* property when the column bindings or fields in the dataset change.

```
procedure BeginLayout;
```

Description Applications cannot call this protected method. Data-aware grids call *BeginLayout* internally before making changes to the *Columns* property or any of the individual *TColumn* objects in *Columns*. Once the change is complete, the data-aware grid calls *EndLayout*. If the response to the change must be cancelled or postponed, the grid calls *CancelLayout*.

Each call to *BeingLayout* increments the *LayoutLock* property. Each call to *EndLayout* or *CancelLayout* decrements the *LayoutLock* property. If *LayoutLock* returns to 0 in a call to *EndLayout*, the *Fields* property is updated to match both the bindings of the *Columns* property and the fields available in the dataset. The *Columns* property is then updated in turn to match the *Fields* property. This ensures that the *Columns* property matches both the specified column bindings and the fields available in the dataset.

See also *AcquireLayoutLock* method, *CancelLayout* method, *Columns* property, *DeferLayout* method, *DefineFieldMap* method, *EndLayout* method, *Fields* property, *LayoutLock* property

BeginUpdate

BeginUpdate increments the *UpdateLock* property when the data in the grid changes.

```
procedure BeginUpdate;
```

Description Applications cannot call this protected method. Data-aware grids call *BeginUpdate* internally before making changes that affect the data that is drawn in the grid. Once the change is complete, the data-aware grid calls *EndUpdate*, which decrements the *UpdateLock* property.

While *UpdateLock* is greater than 0, the grid does not repaint its cells. *BeginUpdate* and *EndUpdate* prevent the grid from flickering when the grid is resized or the user scrolls through the data.

See also *EndUpdate* method, *UpdateLock* property

CancelLayout

CancelLayout decrements the *LayoutLock* property.

```
procedure CancelLayout;
```

Description Applications cannot call this protected method. Data-aware grids call *CancelLayout* internally when postponing the recomputation of *Columns*.

Each call to *BeingLayout* increments the *LayoutLock* property. Each call to *EndLayout* or *CancelLayout* decrements the *LayoutLock* property. If *LayoutLock* returns to 0 in a call to *CancelLayout*, the *Columns* property is not updated. *CancelLayout* is called from the *DeferLayout* method to decrement the *LayoutLock* and prevent the recomputation of *Columns* until a more appropriate time.

See also *BeginLayout* method, *Columns* property, *DeferLayout* method, *EndLayout* method, *LayoutLock* property

CanEditAcceptKey

CanEditAcceptKey indicates whether the *Key* parameter can be entered into the current cell.

```
function CanEditAcceptKey(Key: Char): Boolean; override;
```

Description Data-aware grids use *CanEditAcceptKey* internally to determine whether keys typed by the user are valid for the field represented by the current cell. *CanEditAcceptKey* calls the *IsValidChar* method of the field component associated with the column that has input focus.

CanEditAcceptKey provides a crude validation of keyboard input. More refined validation is provided when the value in the cell is written to the record in the dataset, or by an edit mask.

See also *GetColField* method, *GetEditMask* method, *TField.IsValidChar* method

CanEditModify

CanEditModify indicates whether the current column of the grid can be edited.

```
function CanEditModify: Boolean; override;
```

Description Data-aware grids use *CanEditModify* internally to determine whether to allow the user to edit the current cell. *CanEditModify* returns *True* if the field in the current column can be edited. *CanEditModify* returns *False* if the field cannot be edited. If the dataset is not in edit mode, *CanEditModify* will attempt to put it in edit mode.

Note *CanEditModify* returns False when the current field is a binary large object (BLOB) field. BLOB fields can only be edited through the clipboard or by streaming.

See also *ReadOnly* property, *TColumn.ReadOnly* property, *TField.CanModify* property

CanEditShow

CanEditShow indicates whether the in-place edit control can be created to allow editing.

```
function CanEditShow: Boolean; override;
```

Description *TCustomDBGrid* overrides the inherited method to prevent the in-place edit control from coming up while the layout of the grid is being recomputed.

Grid controls call *CanEditShow* internally before creating the editor for a cell. *CanEditShow* is called after the grid has determined that the field can be edited by calling *CanEditModify*.

See also *CanEditModify* method, *TCustomGrid.CanEditShow* method

CellClick

CellClick calls the *OnCellClick* event handler.

```
procedure CellClick(Column: TColumn); dynamic;
```

Description Applications cannot directly call *CellClick*. It is exposed as a protected method so that descendants can call the *OnCellClick* event handler in response to other events or override the response when the user releases the mouse in a cell of the grid.

See also *OnCellClick* event

ColEnter

ColEnter calls the *OnColEnter* event handler.

```
procedure ColEnter; dynamic;
```

Description Applications cannot directly call *ColEnter*. It is exposed as a protected method so that descendants can call the *OnCellEnter* event handler in response to other events or override the response when a cell in the grid is selected.

See also *OnColEnter* event

ColExit

ColExit calls the *OnColExit* event handler.

```
procedure ColExit; dynamic;
```

Description Applications cannot directly call *ColExit*. It is exposed as a protected method so that descendants can call the *OnColExit* event handler in response to other events or override the response when a cell in the grid loses focus.

See also *OnColExit* event

ColumnMoved

ColumnMoved calls the *OnColumnMoved* event handler.

```
procedure ColumnMoved; dynamic;
```

Description Applications cannot directly call *ColumnMoved*. It is exposed as a protected method so that descendants can call the *OnColumnMoved* event handler in response to other events or override the response after a column in the grid is moved.

See also *OnColumnMoved* event

ColWidthsChanged

ColWidthsChanged updates the column widths in the *Columns* property when the user resizes the columns.

```
procedure ColWidthsChanged; override;
```

Description Applications cannot directly call *ColWidthsChanged*. It is called automatically when the user resizes the columns in the grid. *ColWidthsChanged* overrides the inherited method to update the column widths in the *Columns* property.

See also *Columns* property, *TColumn.Width* property

Create

Create creates and initializes an instance of *TCustomDBGrid*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* for a descendant of *TCustomDBGrid* to create a data-aware grid at runtime. Grids placed on a form at design time are created automatically.

After calling the inherited constructor, *Create* creates the helper objects used by the data-aware grid and initializes

- *Color* to *clWindow*, and *ParentColor* to *False*.
- *DefaultDrawing* to *True*.
- *Options* to [*dgEditing*, *dgTitles*, *dgIndicator*, *dgColumnResize*, *dgColLines*, *dgRowLines*, *dgTabs*, *dgConfirmDelete*, *dgCancelOnExit*].

See also *DefaultDrawing* property, *Options* property, *ParentColor* property, *TControl.Color* property, *TCustomGrid.Create* method

CreateColumns

CreateColumns creates the *TDBGridColumns* object that stores the column descriptors for the grid.

```
function CreateColumns: TDBGridColumns; dynamic;
```

Description Applications cannot call *CreateColumns*. It is exposed as a protected method so that component writers can substitute a descendant of *TDBGridColumns* to manage the column descriptors of a data-aware grid, and so that helper objects in the DBGrids unit can create a set of column descriptors for a grid.

See also *Columns* property

CreateEditor

CreateEditor creates the in-place editor that allows the cells of the grid to be edited.

```
function CreateEditor: TInplaceEdit; override;
```

Description *CreateEditor* overrides the inherited method to create a data-aware descendant of *TInplaceEdit*.

See also *TInplaceEdit* object

CreateWnd

CreateWnd creates the window used to display the data-aware grid.

```
procedure CreateWnd; override;
```

Description *CreateWnd* overrides the inherited method to set up the relationship between the visual display and the data source after the window is created.

See also *TWinControl.CreateWnd* method

DataToRawColumn

DataToRawColumn converts a column index in the *Columns* property to the index of the corresponding column in the visual display.

```
function DataToRawColumn(ACol: Integer): Integer;
```

Description Applications cannot call this protected method. It is used internally to allow the data-aware grid to adjust to the presence or absence of the column that contains the row indicator.

See also *Columns* property, *IndicatorOffset* property, *RawToDataColumn* method

DefaultDrawColumnCell

DefaultDrawColumnCell draws the text in a column cell.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);

procedure DefaultDrawColumnCell(const Rect: TRect; DataCol: Integer; Column: TColumn;
    State: TGridDrawState);
```

Description Call *DefaultDrawColumnCell* from an *OnDrawColumnCell* event handler to lookup up the text representation of a field and write it to the cell. *DefaultDrawColumnCell* does exactly what the grid does when the *DefaultDrawing* property is *True*, except it never draws a focus rectangle on selected cells.

The *Rect* parameter is the position of the cell on the canvas. The *DataCol* parameter is the index of the column in the *Columns* array. The *Column* parameter is the *TColumn* object that describes the display attributes and field binding for the cell. The *State* parameter indicates whether the cell has input focus, whether the cell is selected, and whether the cell is a fixed (non-data) cell such as a column header.

See also *Columns* property, *DefautDrawing* property, *OnDrawColumnCell* event, *TColumn* object

DefaultDrawDataCell

DefaultDrawDataCell draws the text in a column cell.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);

procedure DefaultDrawDataCell(const Rect: TRect; Field: TField; State: TGridDrawState);
```

Description Call *DefaultDrawDataCell* from an *OnDrawDataCell* event handler to lookup up the text representation of a field and write it to the cell. Unlike what the grid draws when the *DefaultDrawing* property is *True*, *DefaultDrawDataCell* gets the alignment for the text from the field component rather than the column descriptor, and it never draws a focus rectangle on selected cells.

The *Rect* parameter gives the position of the cell on the canvas. The *Field* parameter gives the field component that provides the data and alignment

for the cell. The *State* parameter indicates whether the cell has input focus, whether the cell is selected, and whether the cell is a fixed (non-data) cell such as a column header.

See also *DefaultDrawing* property, *OnDrawDataCell* event, *TField* object

DefaultHandler

DefaultHandler brings up the popup menu when a column is right-clicked.

```
procedure DefaultHandler(var Msg); override;
```

Description *DefaultHandler* is the last method to receive Windows messages that are sent to the grid. After calling the inherited method and allowing it to process the message, *DefaultHandler* checks if the *Msg* parameter indicates the right mouse button was released over a column with a popup menu. If the column has a popup menu with its *AutoPopup* property set to *True*, *DefaultHandler* displays the popup menu.

When overriding *DefaultHandler*, be sure to call the inherited method for all unhandled messages.

See also *TColumn.PopupMenu* property, *TPopupMenu.AutoPoppup* property, *TWinControl.DefaultHandler* method

DeferLayout

DeferLayout decrements the *LayoutLock* property and posts a message to the grid to trigger a new layout sequence at a more appropriate time.

```
procedure DeferLayout;
```

Description Applications cannot call this protected method. Data-aware grids call *DeferLayout* internally to postpone the recomputation of the *Columns* property after the dataset changes when it is inappropriate to recompute columns immediately.

DeferLayout calls *CancelLayout* to decrement the *LayoutLock* property which was incremented in response to the change in the dataset. It then posts a message to the grid which will trigger another layout attempt at a later point in time. When the message is received, the grid calls *BeginLayout* and then either *EndLayout* or *DeferLayout*, depending on whether it is an appropriate time to proceed with the computation of *Columns*.

See also *BeginLayout* method, *CancelLayout* method, *Columns* property, *EndLayout* method, *LayoutLock* property

DefineFieldMap

DefineFieldMap builds the *Fields* array from the *Columns* property and the data source.

```
procedure DefineFieldMap; virtual;
```

Description Applications cannot call this protected method. *DefineFieldMap* rebuilds the *Fields* array to reflect the fields from the data source that correspond to the columns in the *Columns* property. After calling *DefineFieldMap*, the *Columns* property should be updated to use the field components in the *Fields* array. This ensures that the *Columns* property reflects any changes imposed by the dataset.

See also *Columns* property, *EndLayout* method, *Fields* property

Destroy

Destroy destroys an instance of *TCustomDBGrid*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the data-aware grid is not already freed, and only then calls *Destroy*. *Destroy* frees all the helper objects used by the grid.

See also *Free* method

DrawCell

DrawCell draws the cell specified by the *ACol* and *ARow* parameters.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState); override;
```

Description *DrawCell* overrides the inherited method to use the *Columns* property when drawing cells in the grid.

DrawCell sets up the font and brush and paints the background of the cell. If the *DefaultDrawing* property is *True*, *DrawCell* fetches the data for the cell and writes it to the position described by the *ARect* parameter. Whether *DefaultDrawing* is *True* or *False*, *DrawCell* generates the *OnDrawDataCell* and *OnDrawColumnCell* events. Finally, if *DefaultDrawing* is *True* and either the cell has focus or the cell is selected and *Options* includes *dgAlwaysShowSelection*, *DrawCell* draws a focus rectangle around the cell.

See also *Columns* property, *DefaultDrawing* property, *OnDrawColumnCell* event, *OnDrawDataCell* event, *Options* property

DrawColumnCell

DrawColumnCell calls the *OnDrawColumnCell* event handler.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
procedure DrawColumnCell(const Rect: TRect; DataCol: Integer; Column: TColumn;
    State: TGridDrawState); dynamic;
```

Description Applications cannot call *DrawColumnCell* directly. It is exposed as a protected method so that descendants can call the *OnDrawColumnCell* event handler or override the method to block the event.

See also *OnDrawColumnCell* event

DrawDataCell

DrawDataCell calls the *OnDrawDataCell* event handler.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
procedure DrawDataCell(const Rect: TRect; Field: TField; State: TGridDrawState); dynamic;
```

Description Applications cannot call *DrawDataCell* directly. It is exposed as a protected method so that descendants can call the *OnDrawDataCell* event handler or override the method to block the event.

See also *OnDrawDataCell* event

EditButtonClick

EditButtonClick calls the *OnEditButtonClick* event handler.

```
procedure EditButtonClick; dynamic;
```

Description Applications cannot call *EditButtonClick* directly. It is exposed as a protected method so that descendants can call the *OnEditButtonClick* event handler or override the method to block the event.

See also *OnEditButtonClick* event

EndLayout

EndLayout decrements the *LayoutLock* property and recomputes the *Columns* property if *LayoutLock* reaches 0.

```
procedure EndLayout;
```

Description Applications cannot call this protected method. Data-aware grids call *Endlayout* internally after changes to the *Columns* property or the dataset. *EndLayout* decrements the *LayoutLock* property which was previously incremented by *BeingLayout* before the *Columns* property was changed or when notification of the change to the dataset was received.

If *LayoutLock* returns to 0, *EndLayout* updates the *Fields* property to match both the bindings of the *Columns* property and the fields available in the dataset. The *Columns* property is then updated in turn to match the *Fields* property. This ensures that the *Columns* property matches both the specified column bindings and the fields available in the dataset.

See also *BeginLayout* method, *Columns* property, *DefineFieldMap* method, *Fields* property, *LayoutLock* property

EndUpdate

EndUpdate decrements the *UpdateLock* property.

```
procedure EndUpdate;
```

Description Applications cannot call this protected method. Data-aware grids call *EndUpdate* internally after making changes that affect the data that is drawn in the grid. *EndUpdate* returns the *UpdateLock* to the state it was in before the *BeginUpdate* method was called at the start of the changes.

While *UpdateLock* is greater than 0, the grid does not repaint its cells. When *UpdateLock* returns to 0, painting is re-enabled. *BeginUpdate* and *EndUpdate* prevent the grid from flickering when the grid is resized or the user scrolls through the data.

See also *BeginUpdate* method, *UpdateLock* property

GetColField

GetColField returns the field component that represents the binding of a specified column.

```
function GetColField(DataCol: Integer): TField;
```

Description Call *GetColField* to obtain the field component for a column in the data-aware grid. The *DataCol* parameter is the index of the column in the *Columns* property, where 0 is the first column, 1 is the second column, and so on.

Unlike the *Fields* property, *GetColField* will return field components for columns that do not represent fields in the dataset, such as calculated fields.

See also *Columns* property, *Fields* property

C

GetEditLimit

GetEditLimit limits the number of characters that can be typed into cells representing string fields.

```
function GetEditLimit: Integer; override;
```

Description *GetEditLimit* is called by the in-place editor to determine if a limit should be placed on the number of characters a user can type into a cell. *GetEditLimit* sets no limits except when the selected cell represents a string field.

See also *CreateEditor* method

GetEditMask

GetEditMask returns the input mask associated with the field for the selected cell.

```
function GetEditMask(ACol, ARow: Longint): string; override;
```

Description The in-place editor calls *GetEditMask* to obtain any input mask associated with the field binding of the indicated cell. *GetEditMask* returns the *EditMask* property of the field component from the *Columns* array.

The *ACol* and *ARow* parameters indicate the index of the row and column of the cell in the grid. The *ACol* parameter is not the same as the index of the column in the *Columns* array if the *Options* property includes dgIndicator.

See also *Columns* property, *Options* property, *RawToDataColumn* method, *TField.EditMask* property

GetEditText

GetEditText returns the value of the indicated cell formatted for editing.

```
function GetEditText(ACol, ARow: Longint): string; override;
```

Description The in-place editor calls *GetEditText* to obtain the text for the indicated cell. *GetEditText* locates the field component for the cell from the *Columns* array and returns its value formatted for editing.

The *ACol* and *ARow* parameters indicate the index of the row and column of the cell in the grid. The *ACol* parameter is the "raw" column index, which is not the same as the index of the column in the *Columns* array if the *Options* property includes *dgIndicator*.

Note The string returned by *GetEditText* may differ from the string returned by the *GetFieldValue* method if the field formats its value differently for editing and display.

See also *Columns* property, *Options* property, *RawToDataColumn* method, *TField.EditMask* property

GetFieldValue

GetFieldValue returns the value of the indicated cell formatted for display.

```
function GetFieldValue(ACol: Integer): string;
```

Description Call *GetFieldValue* to obtain the text that should be displayed in a cell. For example, an *OnDrawColumnCell* event handler can use *GetFieldValue* to obtain the text for the cell to draw it in a special way, rather than calling *DefaultDrawColumnCell*.

The *ACol* parameter is the index of the column in the columns array. If the *Options* property includes *dgIndicator*, this is not the same as the index of the column in the physical display.

See also *Columns* property, *OnDrawColumnCell* event, *Options* property, *TField.DisplayText* property

HighlightCell

HighlightCell indicates whether the cell indicated by the *DataCol* and *DataRow* parameters should be highlighted when it is drawn.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);

function HighlightCell(DataCol, DataRow: Integer; const Value: string;
    AState: TGridDrawState): Boolean; virtual;
```

Description Applications cannot call this protected method. It is called internally by the *DrawCell* method to determine whether the indicated cell should be highlighted. *HighlightCell* returns *True* if the cell has focus, the cell is selected and *Options* includes *dgAlwaysShowSelection*, or the cell is in a selected row and *Options* includes *dgRowSelect*.

See also *DrawCell* method, *Options* property

KeyDown

KeyDown provides special processing when the user presses down on a key.

```
procedure KeyDown(var Key: Word; Shift: TShiftState); override;
```

Description Applications cannot call *KeyDown*. It is called automatically when the user presses a key, before the *KeyPress* method is called.

KeyDown generates an *OnKeyDown* event and then responds to the following keystrokes:

Keystroke	Response
UpArrow	Move to the previous row. If *Options* includes *dgRowSelect*, select it.
Ctrl+UpArrow	Move to the first row in the grid. Deselect any selected rows.
DownArrow	Move to the next row. If *Options* includes *dgRowSelect*, select it.
Ctrl+DownArrow	Move to the last row in the grid. Deselect any selected rows.
PageUp	Move up by the number of records shown in the grid. Deselect any selected rows.
Ctrl+PageUp	Move to the first row in the grid. Deselect any selected rows.
PageDown	Move down by the number of records shown in the grid. Deselect any selected rows.
LeftArrow	Move left one column. If *Options* includes *dgRowSelect*, move to the previous row without selecting it.
Ctrl+LeftArrow	Move to the first column in the current row.
RightArrow	Move right one column. If *Options* includes *dgRowSelect*, move to the next row without selecting it.
Ctrl+RightArrow	Move to the last column in the current row.
Home	Move to the first column of the current row. If *Options* includes *dgRowSelect* or the grid has only one column, move to the first record in the dataset. Deselect any selected rows.
Ctrl+Home	Move to the first record in the dataset. Deselect any selected rows.

Keystroke	Response
End	Move to the last column of the current row. If *Options* includes *dgRowSelect* or the grid has only one column, move to the last record in the dataset. Deselect any selected rows.
Ctrl+End	Move to the last record in the dataset. Deselect any selected rows.
Tab	Move to the next cell in the grid.
Shift+Tab	Move to the previous cell in the grid.
Ctrl+Delete	Delete the current selection.
Insert	Insert a new row above the current one, and move to it. Deselect any selected rows.
Escape	Cancel any unposted edits. Deselect any selected rows.
F2	Put the grid in edit mode.

See also *KeyPress* method, *Options* property, *TWinControl.OnKeyDown* event

KeyPress

KeyPress overrides the inherited method to post edits to the database when the user presses *Return*.

```
procedure KeyPress(var Key: Char); override;
```

Description When the user presses a key, first the *KeyDown* method is called, then the *KeyPress* method, and finally the *KeyUp* method when the user releases the key. *KeyPress* writes any pending edits to the database and calls the inherited *KeyPress* method.

See also *KeyDown* method, *TCustomGrid.KeyPress* method, *TWinControl.KeyUp* method

LayoutChanged

LayoutChanged recomputes the *Columns* property.

```
procedure LayoutChanged; virtual;
```

Description Applications cannot call this protected method. It is called automatically in response to changes the affect the layout of the columns. These include

- Changes to the fields in the dataset.
- Changes in the *Options* property that affect the layout.
- Changes to the *TitleFont* property.

LayoutChanged calls *AcquireLayoutLock* to initiate the changes to *Columns*. If *AcquireLayoutLock* does not indicate that the call to *LayoutChanged* is recursive, *LayoutChanged* calls *EndLayout* to perform the actual computation.

See also *AcquireLayoutLock* method, *Columns* property, *EndLayout* method, *Options* property, *TitleFont* property

LinkActive

LinkActive recomputes the *Columns* property and the vertical scrollbar of the grid.

```
procedure LinkActive(Value: Boolean); virtual;
```

Description Applications cannot call this protected method. It is called automatically when the dataset is opened. *LinkActive* calls *LayoutChanged* to recompute the *Columns* property, and adjusts the vertical scrollbar for the data in the dataset.

See also *Columns* property, *LayoutChanged* method

Loaded

Loaded sets up the *Columns* property once the data grid and all its child components have been loaded.

```
procedure Loaded; override;
```

Description Applications cannot call this protected method. *Loaded* overrides the inherited method to lay out the grid once all of its parts have been loaded.

See also *Columns* property

MouseDown

MouseDown provides special processing when the user clicks on the mouse.

```
procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); override;
```

Description *MouseDown* is called automatically when the user presses the mouse button. If the mouse is pressed in the row or column resize region of the grid, *MouseDown* posts any pending edits and calls the inherited method to handle the resize. When *Options* includes *dgRowSelect*, *MouseDown* selects the row that received the click, or toggles its selection state if the *Ctrl* key is pressed. If the mouse is clicked in a data cell, *MouseDown* tries to put the grid into edit mode.

See also *MouseUp* method, *Options* property, *TCustomGrid.MouseDown* method

MouseUp

MouseUp provides special processing when the user releases the mouse button.

```
procedure MouseUp(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); override;
```

Description *MouseUp* is called automatically when the user releases the mouse button. After calling the inherited method, *MouseUp* locates the cell which was under the mouse when it was released. *MouseUp* generates an *OnTitleClick* or *OnCellClick* event, depending on the location of the mouse.

See also *OnCellClick* event, *OnTitleClick* event, *TCustomGrid.MouseUp* method

Notification

Notification adjusts the grid when any of its child components are destroyed.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description After calling the inherited method, *Notification*

- Checks if the pop-up menu for any of its columns is going away, and if so, resets the *PopupMenu* property of the column object.
- Checks if the data source is going away, and if so, sets the *DataSource* property to **nil**.
- Checks if any of the field components associated with its columns is going away, and if so, removes the column from the layout.

See also *Columns* property, *DataSource* property, *TColumn.PopupMenu* property

RawToDataColumn

RawToDataColumn converts the index of a column in the visual display to the corresponding index in the *Columns* property.

```
function RawToDataColumn(ACol: Integer): Integer;
```

Description Applications cannot call this protected method. It is used internally to allow the data-aware grid to adjust to the presence or absence of the column that contains the row indicator.

See also *Columns* property, *DataToRawColumn* method, *IndicatorOffset* property

Scroll

Scroll updates the image of the grid to reflect a new record position.

```
procedure Scroll(Distance: Integer); virtual;
```

Description *Scroll* is called in response to changes in the current record in the dataset. *Scroll* updates the image in the grid, scrolling the data if necessary.

See also *TimedScroll* method

SetColumnAttributes

SetColumnAttributes sets the column widths and disables tabbing to cells that can't be edited.

```
procedure SetColumnAttributes; virtual;
```

Description Applications cannot call this protected method. It is called automatically when the *Columns* property is recomputed, to adjust the column widths and ensure that the user can only tab to fields that can be edited.

See also *Columns* property, *EndLayout* method

SetEditText

SetEditText updates the edited version of the current cell's data.

```
procedure SetEditText(ACol, ARow: Longint; const Value: string); override;
```

Description *SetEditText* is called by the in-place editor so that the grid can store the new value for the current cell. The grid stores this value until the user moves to a new cell or presses *Escape*. If the user moves to a new cell, this value is written to the record in the database. If the user presses *Escape*, the value is discarded.

See also *GetEditText* method

StoreColumns

StoreColumns indicates whether the columns of the grid are stored when the grid is saved, or whether they are generated dynamically.

```
function StoreColumns: Boolean;
```

Description *StoreColumns* causes the *Columns* property to be stored only when the *Columns* property has a *State* property of *csCustomized*. The *State* property for *Columns* indicates whether the columns are generated automatically from the fields in the dataset, or if they are specified programmatically or by the Columns editor.

See also *Columns* property, *TDBGridColumns.State* property

C

TimedScroll

TimedScroll moves the current record in response to the user dragging the mouse outside of the data cells.

```
type TGridScrollDirection = set of (sdLeft, sdRight, sdUp, sdDown);
```

```
procedure TimedScroll(Direction: TGridScrollDirection); override;
```

Description If the *Direction* parameter includes *sdUp*, *TimedScroll* moves the dataset to the first visible record in the grid. If the *Direction* parameter includes *sdDown*, *TimedScroll* moves the dataset to the last visible record in the grid. After adjusting the current record, *TimedScroll* calls the inherited method to update the image.

See also *TCustomGrid.TimedScroll* method

TitleClick

TitleClick calls the *OnTitleClick* event handler.

```
procedure TitleClick; dynamic;
```

Description Applications cannot call *TitleClick* directly. It is exposed as a protected method so that descendants can call the *OnTitleClick* event handler or override the method to block the event.

See also *OnTitleClick* event

ValidFieldIndex

ValidFieldIndex indicates whether an index into the columns array refers to a valid field in the dataset.

```
function ValidFieldIndex(FieldIndex: Integer): Boolean;
```

Description Call *ValidFieldIndex* to determine whether a column displays data from a field in the dataset. *ValidFieldIndex* returns *True* if the column is associated with a database field. *ValidFieldIndex* returns *False* if the column is a calculated field or if the field for the column is undefined.

See also *Columns* property, *Fields* property, *GetColField* method

TCustomDecisionGraph

Unit: MXGraph

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomPanel • TCustomTeePanel • TCustomChart • TChart • TCustomDecisionGraph

TCustomDecisionGraph is the base class for controls that graph crosstabulated data.

Description Use *TCustomDecisionGraph* as a base class when defining objects that graph the data from a decision source. Decision graphs chart a single summary that is calculated over a number of dimensions. The dimensions appear as axes of the graph.

The field that is summarized and the fields (dimensions) over which it is summarized can be changed dynamically at runtime to view any configuration of summary and dimensions chosen from the dimension map of a decision cube. Changes to the dimensions and summary can be initiated using a *TDecisionPivot* object.

Do not create instances of *TCustomDecisionGraph*. To put a decision graph on a form, use *TDecisionGraph*.

See also *TDecisionCube* object, *TDecisionGraph* object, *TDecisionPivot* object, *TDecisionSource* object

Properties

The following table lists the properties of *TCustomDecisionGraph*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
AllowPanning	published	pmBoth	TChart
AllowZoom	published	True	TChart
AnimatedZoom	published	False	TChart
AnimatedZoomSteps	published	8	TChart
AxisVisible	published	True	TChart (TCustomChart)
BackColor	published	clTeeColor	TChart (TCustomChart)
BackImage	published		TChart
BackImageInside	published	False	TChart
BackImageMode	published	pbmStretch	TChart
BevelInner	published	bvNone	TCustomPanel
BevelOuter	published	bvRaised	TCustomPanel
BevelWidth	published	1	TCustomPanel
BorderStyle	published	bsNone	TCustomPanel
BorderWidth	published	0	TCustomPanel
BottomAxis	published		TChart (TCustomChart)
BottomWall	published		TChart
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
BufferedDisplay	public		TChart (TCustomChart)
Canvas	public, read-only		TCustomControl
Chart3DPercent	published	Def3DPercent	TChart (TCustomChart)
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ClipPoints	published	True	TChart (TCustomChart)
Color	published	clBtnFace	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl

Property	Visibility	Default	Documented in
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DecisionSource	public		TCustomDecisionGraph
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Foot	published		TChart
Frame	published		TChart (TCustomChart)
Gradient	published		TChart
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
LeftAxis	published		TChart (TCustomChart)
LeftWall	published		TChart
Legend	published		TChart (TCustomChart)
Locked	published	False	TCustomPanel
MarginBottom	published	TeeDefVerticalMargin	TChart (TCustomTeePanel)
MarginLeft	published	TeeDefHorizMargin	TChart (TCustomTeePanel)
MarginRight	published	TeeDefHorizMargin	TChart (TCustomTeePanel)
MarginTop	published	TeeDefVerticalMargin	TChart (TCustomTeePanel)
MaxPointsPerPage	published	0	TChart (TCustomChart)
Monochrome	published	False	TChart (TCustomChart)
Name	published		TControl
Owner	public, read-only		TComponent
Page	published	1	TChart (TCustomChart)
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
PrintResolution	public	0	TChart
RightAxis	published		TChart (TCustomChart)
SaveSeries	public		TCustomDecisionGraph
ScaleLastPage	published	True	TChart (TCustomChart)
Series	public, read-only		TChart (TCustomChart)
SeriesList	published, read-only		TChart (TCustomChart)
ShowHint	published		TControl

Property	Visibility	Default	Documented in
Showing	public, read-only		TWinControl
TabOrder	published	–1	TWinControl
TabStop	published	False	TWinControl
Tag	published	0	TComponent
Title	published		TChart
Top	published		TControl
TopAxis	published		TChart (TCustomChart)
VCLComObject	public		TComponent
View3d	published	True	TChart (TCustomChart)
View3dWalls	published	True	TChart (TCustomChart)
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl
Zoomed	public		TChart

DecisionSource

DecisionSource is the decision source object that supplies the data values for the decision graph.

property DecisionSource: TDecisionSource;

Description Set *DecisionSource* to link the decision graph to the crosstabulated data from a decision cube. The *DecisionSource* object maintains information about the current pivot state of the data. Thus, any two decision controls that share the same decision source will display the data from the decision cube using the same configuration of dimensions, including any closed dimensions or drilled down values.

See also *TDecisionCube* object, *TDecisionSource* object

SaveSeries

SaveSeries specifies whether the *Series* property should be saved with the decision graph.

property SaveSeries: Boolean;

Description Set *SaveSeries* to *True* to save any series that have been customized using the graph property dialog. Saving series can be memory intensive. Set *SaveSeries* to *False* to skip this operation.

See also *Series* property

Events

The following table lists the events of *TCustomDecisionGraph*:

Event	Visibility	Documented in
OnAfterDraw	published	TChart (TCustomChart)
OnAllowScroll	published	TChart
OnClick	published	TControl
OnClickAxis	published	TChart
OnClickBackground	published	TChart
OnClickLegend	published	TChart
OnClickSeries	published	TChart
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnGetAxisLabel	published	TChart (TCustomChart)
OnGetLegendPos	published	TChart
OnGetLegendRect	published	TChart
OnGetLegendText	published	TChart (TCustomChart)
OnGetNextAxisLabel	published	TChart (TCustomChart)
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnPageChange	published	TChart (CustomChart)
OnResize	published	TChart (TCustomPanel)
OnScroll	published	TChart
OnUndoZoom	published	TChart
OnZoom	published	TChart

Methods

The following table lists the methods of *TCustomDecisionGraph*:

Method	Visibility	Documented in
ActiveSeriesLegend	public	undocumented
AddSeries	public	TChart (TCustomChart)
Assign	public	TPersistent
BackWallRect	public	undocumented
BeginDrag	public	TControl
BringToFront	public	TControl

Method	Visibility	Documented in
Broadcast	public	TWinControl
CalcClickedPart	public	undocumented
CalcMetaBounds	public	undocumented
CalcSize3d	public	TChart (TCustomChart)
CalcSize3dWalls	public	TChart (TCustomChart)
CanFocus	public	TWinControl
CanvasChanged	public	undocumented
ChartPrintRect	public	TChart
ChartRegionRect	public	TChart (TCustomChart)
CheckDatasource	public	TChart (TCustomChart)
CheckPenWidth	public	undocumented
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ClipRectangle	public	undocumented
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
CopyToClipboardBitmap	public	undocumented
CopyToClipboardMetafile	public	undocumented
Create	public	TCustomDecisionGraph
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCustomDecisionGraph
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DoHorizLine	public	undocumented
DoLine	public	undocumented
DoPolygon	public	undocumented
DoRectangle	public	TChart (TCustomTeePanel)
DoVertLine	public	undocumented
DragDrop	public	TControl
Dragging	public	TControl
Draw	public	TChart (TCustomChart)
DrawToMetaCanvas	public	undocumented
EnableAlign	public	TWinControl
EndDrag	public	TControl

Method	Visibility	Documented in
ExchangeSeries	public	undocumented
FieldAddress	public	TObject
FillSeriesSourceItems	public	undocumented
FillValueSourceItems	public	undocumented
FindComponent	public	TComponent
Focused	public	TWinControl
FontCanvas	public	TChart (TCustomTeePanel)
FormattedLegend	public	TChart (TCustomChart)
FormattedValueLegend	public	TChart (TCustomChart)
Free	public	TObject
FreeAllSeries	public	undocumented
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetASeries	public	TChart (TCustomChart)
GetAxisSeries	public	TChart (TCustomChart)
GetCursorPos	public	TChart (TCustomTeePanel)
GetEditorClass	public	TChart
GetFreeSeriesColor	public	TChart (TCustomChart)
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetLabelsSeries	public	TChart (TCustomChart)
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetRectangle	public	undocumented
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
GetWidthHeight	public	undocumented
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
InternalDraw	public	undocumented
Invalidate	public	TWinControl
IsFreeSeriesColor	public	TChart (TCustomChart)

Method	Visibility	Documented in
IsScreenHighColor	public	TChart
IsValidDataSource	public	TChart (TCustomChart)
MarkText	public	TChart (TCustomChart)
MaxMarkWidth	public	TChart (TCustomChart)
MaxTextWidth	public	TChart (TCustomChart)
MaxXValue	public	TChart (TCustomChart)
MaxYValue	public	TChart (TCustomChart)
MethodAddress	public	TObject
MethodName	public	TObject
MinXValue	public	TChart (TCustomChart)
MinYValue	public	TChart (TCustomChart)
NewInstance	public	TObject
NextPage	public	TChart
NumPages	public	TChart (TCustomChart)
PaintTo	public	TWinControl
Perform	public	TControl
PreviousPage	public	TChart
Print	public	TChart
PrintLandscape	public	TChart
PrintOrientation	public	TChart
PrintPartial	public	TChart
PrintPartialCanvas	public	TChart
PrintPortrait	public	TChart
PrintRect	public	TChart
Realign	public	TWinControl
ReCalcWidthHeight	public	TChart (TCustomTeePanel)
Refresh	public	TControl
RemoveAllSeries	public	TChart (TCustomChart)
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
RemoveSeries	public	TChart (TCustomChart)
Repaint	public	TWinControl
RotateLabel	public	TChart (TCustomTeePanel)
SafeCallException	public	TComponent
SaveToBitmapFile	public	TChart
SaveToMetafile	public	TChart
SaveToMetafileEnh	public	undocumented
SaveToMetafileRect	public	undocumented
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl

Method	Visibility	Documented in
SeriesCount	public	TChart (TCustomChart)
SeriesDown	public	TChart
SeriesTitleLegend	public	undocumented
SeriesUp	public	TChart
SetBooleanProperty	public	undocumented
SetBounds	public	TWinControl
SetBrushCanvas	public	TChart (TCustomTeePanel)
SetColorProperty	public	undocumented
SetDoubleProperty	public	undocumented
SetFocus	public	TWinControl
SetIntegerProperty	public	undocumented
SetInternalCanvas	public	undocumented
SetLongintProperty	public	undocumented
SetStringProperty	public	undocumented
SetTextBuf	public	TControl
Show	public	TControl
TeeCharHeight	public	undocumented
TeeCreateMetafile	public	undocumented
TeeTextHeight	public	TChart (TCustomTeePanel)
TeeTextOut	public	TChart (TCustomTeePanel)
TeeTextWidth	public	TChart (TCustomTeePanel)
UnClipRectangle	public	undocumented
UndoZoom	public	TChart
Update	public	TWinControl
UpdateControlState	public	TWinControl
XLabelText	public	TChart (TCustomChart)
ZoomPercent	public	TChart
ZoomRect	public	TChart

Create

Create creates and initializes an instance of *TCustomDecisionGraph*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create a decision graph at runtime. Decision graphs placed in forms at design time are created automatically. *Create* initializes the *SaveSeries* property to *False*.

See also *SaveSeries* property

Destroy

Destroy destroys an instance of *TCustomDecisionGraph*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the decision grid is not already freed and only then calls *Destroy*.

See also *Free* method

See also *DecisionSource* property, *TDecisionPivot* object

TCustomDecisionGrid

Unit: MXGrid

TObject •TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomGrid • TCustomDecisionGrid

TCustomDecisonGrid is the base class for controls that present crosstabulated data in a two-dimensional grid.

Description Use *TCustomDecisionGrid* as a base class when defining a decision grid object. Decision grids can present a single summary that is calculated over an arbitrary number of dimensions.

The field that is summarized and the fields (dimensions) over which it is summarized can be changed dynamically at runtime to view any configuration of summary and dimensions chosen from the dimension map of a decision cube. Changes to the dimensions and summary can be initiated by direct manipulation of the decision grid, or by using a *TDecisionPivot* object.

Do not create instances of *TCustomDecisionGrid*. To add a decision grid to a form, use the *TDecisionGrid* object.

See also *TDecisionCube* object, *TDecisionGrid* object, *TDecisionPivot* object, *TDecisionSource* object

C

Properties

The following table lists the properties of *TCustomDecisionGrid*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
CaptionColor	protected		TCustomDecisionGrid
CaptionFont	protected		TCustomDecisionGrid
Cells	public, read-only		TCustomDecisionGrid
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl

Property	Visibility	Default	Documented in
ClientWidth	public		TControl
ColCount	protected, read-only		TCustomDecisionGrid
ColWidth	protected		TCustomDecisionGrid
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DataColor	protected		TCustomDecisionGrid
DataFont	protected		TCustomDecisionGrid
DataSumColor	protected		TCustomDecisionGrid
DecisionSource	public		TCustomDecisionGrid
DefaultColWidth	protected		TCustomDecisionGrid
DefaultRowHeight	protected		TCustomDecisionGrid
DesignInfo	public		TComponent
Dimensions	protected		TCustomDecisionGrid
Enabled	public	True	TControl
FixedCols	protected, read-only		TCustomDecisionGrid
FixedRows	protected, read-only		TCustomDecisionGrid
GridLineColor	protected		TCustomDecisionGrid
GridLineWidth	protected		TCustomDecisionGrid
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
LabelColor	protected		TCustomDecisionGrid
LabelFont	protected		TCustomDecisionGrid
LabelSumColor	protected		TCustomDecisionGrid
Left	published		TControl
Name	published		TControl
Options	protected	[cgGridLines, cgOutLiner, cgPivotable]	TCustomDecisionGrid
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl

Property	Visibility	Default	Documented in
RowCount	protected, read-only		TCustomDecisionGrid
RowHeight	protected		TCustomDecisionGrid
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
Totals	public		TCustomDecisionGrid
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

CaptionColor

CaptionColor is the background color of cells containing the names of the row and column dimensions.

property CaptionColor: TColor;

Description Set *CaptionColor* to the color that should be used to make the names of the row and column dimensions stand out. These are the names for the dimensions that are set in the Decision Cube Editor.

The actual values of the fields that make up the row and column dimensions are drawn using the *LabelColor* property.

See also *CaptionFont* property, *DataColor* property, *DataSumColor* property, *Dimensions* property, *LabelColor* property, *LabelSumColor* property, *TColor* type

C

CaptionFont

CaptionFont is the font used to write the names of the row and column dimensions.

property CaptionFont: TFont;

Description Set *CaptionFont* to the font that should be used to make the names of the row and column dimensions stand out. These are the names for the dimensions that are set in the Decision Cube Editor.

The actual values of the fields that make up the row and column dimensions are written using the *LabelFont* property.

See also *CaptionColor* property, *DataFont* property, *Dimensions* property, *LabelFont* property, *TFont* object

Cells

Cells is an indexed array containing the values of all cells in the decision grid.

property Cells[ACol, ARow: Integer]: String;

Description Use *Cells* to obtain the value of a cell in the grid, formatted as a string. The *ACol* and *ARow* parameters indicate the column and row indexes of the cell, where (0,0) indicates the top left cell of the data region (not the top left cell of the grid). Cells that have been closed up using the Display Data or Display SubTotals menu items are still counted even though they are not visible.

If the cell is a row or column caption (negative value for *ACol* or *ARow*), *Cells* returns the name of the dimension. If the cell is a row or column label, *Cells* returns the field value of the dimension field. If the cell is a subtotal label, *Cells* returns the string *Sum*. If the cell is in the data region of the grid, *Cells* returns the summarized value.

See also *CellDrawState* method, *CellValueArray* method

ColCount

ColCount is the total number of columns in the decision grid.

property ColCount:integer;

Description Read *ColCount* to determine the number of columns in the decision grid, including the columns containing dimension captions and label cells. Compute the number of columns in the data region of the decision grid by subtracting the value of *FixedCols* from *ColCount*.

See also *FixedCols* property

ColWidth

ColWidth is the default column width.

property ColWidth: Integer;

Description Set *ColWidth* to the desired width of the columns in the decision grid that are not resized by the user. Setting *ColWidth* changes the size of all columns in the grid, including columns that have been resized using the mouse.

See also *RowHeight* property

DataColor

DataColor is the background color of the cells containing summarized values.

property DataColor: TColor;

Description Set *DataColor* to the color that should be used for the scrollable data region of the decision grid. The data region contains the values of the current

summary of the decision cube, where each cell represents the value of a field summarized over the values indicated by the row and column dimensions.

The subtotals that represent values summarized over all values within a given dimension (as well as the single value in other dimensions) are drawn using the *DataSumColor* property.

See also *CaptionColor* property, *DataFont* property, *DataSumColor* property, *LabelColor* property, *LabelSumColor* property, *TColor* type

DataFont

DataFont is the font used to write the summarized values in the scrollable region of the decision grid.

```
property DataFont: TFont;
```

Description Set DataFont to the font that should be used for the data region of the decision grid. The data region contains the values of the current summary of the decision cube, where each cell represents the value of a field summarized over the values indicated by the row and column dimensions. It also includes the dimension subtotals over the summary field.

See also *CaptionFont* property, *DataColor* property, *LabelFont* property, *TFont* object

DataSumColor

DataSumColor is the background color for cells that represent subtotals over entire dimensions.

```
property DataSumColor: TColor;
```

Description Set *DataSumColor* to make the subtotals that represent an entire dimension stand out from the summary values that summarize information over a single value in each dimension. The subtotal cells are placed in the scrollable data region at the ends of the set of rows or columns that correspond to a given dimension.

See also *CaptionColor* property, *DataColor* property, *DataFont* property, *LabelColor* property, *LabelSumColor* property, *TColor* type, *Totals* property

DecisionSource

DecisionSource provides the crosstabulated data to the decision grid.

```
property DecisionSource: TDecisionSource;
```

Description Set *DecisionSource* to link the decision grid to the crosstabulated data from a decision cube. The *DecisionSource* object maintains information about the current pivot state of the data. Thus, any two decision controls that share the same decision source will display the data from the decision cube using the same configuration of row and column dimensions, including any closed dimensions or drilled down values.

See also *TDecisionCube* object, *TDecisionSource* object

DefaultColWidth

DefaultColWidth is the width of all columns that are not explicitly resized.

property ColWidth: Integer;

Description Set *DefaultColWidth* to the desired width of the columns in the decision grid that are not resized using the mouse. Setting *DefaultColWidth* changes the size of all columns in the grid, including columns that have been resized.

See also *DefaultRowHeight* property

;DefaultRowHeight

DefaultRowHeight is the height of all rows that are not explicitly resized.

property RowHeight: Integer;

Description Set *DefaultRowHeight* to the desired height of the rows in the decision grid that are not resized using the mouse. Setting *DefaultRowHeight* changes the size of all rows in the grid, including rows that have been resized.

See also *DefaultColWidth* property

Dimensions

Dimensions is a collection of *TDisplayDim* objects that control the appearance of each dimension in the decision grid.

property Dimensions: TDisplayDims;

Description Set the properties of the individual *TDisplayDim* objects to control the appearance of individual dimensions within the grid. The *TDisplayDim* objects have properties that can override the various color and font properties of the grid to make individual dimensions stand out. They also allow the dimension names, alignments, and formats to be customized, and permit the dimension subtotals for a dimension to be turned on or off.

Note Do not change the *Dimensions* property, only the properties of individual *TDisplayDim* objects accessed through the *Dimensions* property. To programmatically change the configuration of dimensions that appears in the grid, use the decision source object.

See also *CaptionColor* property, *CaptionFont* property, *DataColor* property, *DataFont* property, *DataSumColor* property, *LabelColor* property, *LabelFont* property, *LabelSumColor* property, *TDecisionSource* object, *TDisplayDim* object, *TDisplayDims* object

FixedCols

FixedCols is the number of columns used for dimension captions and dimension value labels.

```
property FixedCols:integer;
```

Description Read *FixedCols* to determine the number of columns in the decision grid that are not part of the data region. Subtract *FixedCols* from the value of the *ColCount* property to obtain the number of columns in the data region.

See also *ColCount* property

FixedRows

FixedRows is the number of rows used for dimension captions and dimension value labels.

```
property FixedRows:integer;
```

Description Read *FixedRows* to determine the number of rows in the decision grid that are not part of the data region. Subtract *FixedRows* from the value of the *RowCount* property to obtain the number of rows in the data region.

See also *RowCount* property

GridLineColor

GridLineColor is the color of the lines that separate the cells in the decision grid.

```
property GridLineColor: TColor;
```

Description Set *GridLineColor* to specify the color of the lines drawn between the cells of the grid. The grid lines only appear when the *Options* property includes *cgGridLines*.

See also *GridLineWidth* property, *Options* property, *TColor* type

GridLineWidth

GridLineWidth is the width, in pixels, of the lines that separate the cells of the grid.

```
property GridLineWidth: Integer;
```

Description Set *GridLineWidth* to make the lines that separate the cells in the decision grid heavier or lighter. The grid lines only appear when the *Options* property includes *cgGridLines*.

See also *GridLineColor* property, *Options* property

LabelColor

LabelColor is the color of the individual row and column labels.

```
property LabelColor: TColor;
```

Description Set *LabelColor* to the color that should be used to make the row and column labels stand out. The row and column labels of the decision grid contain the values of the fields that correspond to the dimensions of the decision cube. These are the values over which the data is summarized.

See also *CaptionColor* property, *DataColor* property, *DataSumColor* property, *LabelFont* property, *LabelSumColor* property, *TColor* type

LabelFont

LabelFont is the font used to write the individual row and column labels.

```
property LabelFont: TFont;
```

Description Set *LabelFont* to the font that should be used to write the labels that display the values of the dimensions over which data is summarized. The row and column labels of the decision grid contain the values of the fields that correspond to the dimensions of the decision cube. *LabelFont* is also used when writing the labels for rows or columns that represent dimension subtotals.

See also *CaptionFont* property, *DataFont* property, *LabelColor* property, *LabelSumColor* property, *TFont* object

LabelSumColor

LabelSumColor is the background color for the subtotal labels.

```
property LabelSumColor: TColor;
```

Description Set *LabelSumColor* make the subtotal labels stand out from the labels that represent individual field values along a dimension. The subtotal labels appear at the end of the set of field value labels for an individual dimension.

See also *CaptionColor* property, *DataColor* property, *DataSumColor* property, *LabelColor* property, *LabelFont* property, *TColor* type, *Totals* property

Options

Options specifies various settings that affect the entire grid display.

```
type
  TDecisionGridOption = (cgGridLines, cgOutliner, cgPivotable);
  TDecisionGridOptions = set of TDecisionGridOption;

property Options: TDecisionGridOptions;
```

Description Set *Options* to include the desired properties for the grid. *Options* is a set drawn from the following values:

Value	Meaning
cgGridLines	Vertical and horizontal lines are drawn to separate the cells in the scrollable data region.
cgOutliner	The dimension caption cells contain plus (+) and minus (-) signs that open or close dimensions in the grid when clicked. The minus sign closes (hides) an adjacent dimension. The plus sign opens (shows) an adjacent dimension.
cgPivotable	Dimensions can be rearranged by clicking on the dimension caption cells and dragging them to a new position.

Note If *Options* does not include *cgOutliner*, individual dimensions can still be opened and closed using the decision pivot. Similarly, if *Options* does not include *cgPivotable*, individual dimensions can still be opened and closed with the decision pivot.

See also *GridLineColor* property, *GridLineWidth* property, *TDecisionPivot* object

RowCount

RowCount is the total number of rows in the decision grid.

```
property ColCount: Integer;
```

Description Read *RowCount* to determine the number of rows in the decision grid, including the rows containing dimension captions and label cells. Compute the number of rows in the data region of the decision grid by subtracting the value of *FixedRows* from *RowCount*.

See also *FixedRows* property

C

RowHeight

RowHeight is the default row height.

```
property RowHeight: Integer;
```

Description Set *RowHeight* to the desired height of the rows in the decision grid that are not resized by the user. Setting *RowHeight* changes the size of all rows in the grid, including rows that have been resized using the mouse.

See also *ColWidth* property

Totals

Totals indicates whether the grid includes subtotals.

```
property Totals: Boolean;
```

Description Set *Totals* to turn row and column subtotals on or off for the whole grid. To turn subtotals on or off for individual dimensions, use the *Dimensions* property.

See also *Dimensions* property

Events

The following table lists the events of *TDecisionGrid*:

Event	Visibility	Documented in
OnDecisionDrawCell	public	TCustomDecisionGrid
OnDecisionExamineCell	public	TCustomDecisionGrid
OnSelectCell	public	TCustomDecisionGrid
OnTopLeftChanged	public	TCustomDecisionGrid

OnDecisionDrawCell

OnDecisionDrawCell occurs just before a cell in the grid is drawn.

```
type
  TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
  TDecisionDrawStates = (dsGroupStart, dsRowCaption, dsColCaption, dsSum, dsRowValue,
    dsColValue, dsData, dsOpenAfter, dsCloseAfter, dsRowIndicator, dsColIndicator, dsRowPlus,
    dsColPlus, dsNone);
  TDecisionDrawState = set of TDecisionDrawStates;

TDecisionDrawCellEvent = procedure (Sender: TObject; Col, Row: Longint; var Value: string;
  var AFont: TFont; var AColor: TColor; AState: TGridDrawState;
  ADrawState: TDecisionDrawState) of Object;

property OnDecisionDrawCell: TDecisionDrawCellEvent;
```

Description Write an *OnDecisionDrawCell* event handler to change the color, font, or text used to display an individual cell in the grid. Change the *AColor*, *AFont*, or *Value* parameter to change the color, font, or text respectively.

For example, to change the row and column labels from the values of the fields in the dataset to customized labels, change the *Value* parameter of the event handler when the *Row* and *Column* parameters represent a label cell.

The *Row* and *Column* parameters indicate the row and column of a cell within the grid, where (0,0) specifies the upper left summary value, not the upper left corner of the grid. Thus, the label cells contain negative row or column indexes. In addition, the *AState* parameter includes *gdFixed* when the cell is a label cell. *AState* also indicates whether the cell being drawn is selected.

The *ADrawState* parameter indicates the type of cell that is about to be drawn. *ADrawState* includes one or more of the following values:

Value	Meaning
dsGroupStart	The cell is the first row or column label for a dimension. Always appears with *dsRowValue* or *dsColValue*.
dsRowCaption	The cell displays a row caption (the name of a dimension whose values are listed down a column on the left of the grid).
dsColCaption	The cell displays a column caption (the name of a dimension whose values are listed along the top of the grid).
dsSum	The cell displays a subtotal label or value. Always appears with *dsRowValue*, *dsColValue*, or *dsData*.
dsRowValue	The cell is a row label. It represents a value from a dimension listed down a column on the left of the grid or a summary label that appears at the end of each group of dimension values.
dsColValue	The cell is a column label. It represents a value from a dimension listed across the top of the grid or a summary label that appears at the end of the dimension.
dsData	The cell is in the scrollable data region of the grid.
dsOpenAfter	The cell has a plus sign (+) on the right that will open following dimension when clicked. Always appears with *dsRowCaption*, *dsColCaption*, *dsRowPlus*, or *dsColPlus*.
dsCloseAfter	The cell has a minus sign (–) on the right that will close the following dimension when clicked. Always appears with *dsRowCaption*, *dsColCaption*, *dsRowPlus*, or *dsColPlus*.
dsRowIndicator	The cell appears on the left grid margin that is used to display the open and close indicators (+ and –).
dsColIndicator	The cell appears on the upper grid margin that is used to display the open and close indicators (+ and –).
dsRowPlus	The cell appears in the left margin and contains a plus sign (+) indicating that the adjacent dimension is closed. Always appears with *dsRowIndicator*.
dsColPlus	The cell appears in the top margin and contains a plus sign (+) indicating that the adjacent dimension is closed. Always appears with *dsColIndicator*.
dsNone	The cell has no function.

See also *CaptionColor* property, *CaptionFont* property, *DataColor* property, *DataFont* property, *DataSumColor* property, *LabelColor* property, *LabelFont* property, *LabelSumColor* property

OnDecisionExamineCell

OnDecisionExamineCell occurs when the user right-clicks on a cell in the data region of the grid.

```
type TDecisionExamineCellEvent = procedure (Sender: TObject; ICol, IRow: Longint; ISum:
  Integer; const ValueArray: TSmallIntArray) of Object;

property OnDecisionExamineCell: TDecisionExamineCellEvent;
```

Description Write an *OnDecisionExamineCell* event handler to take specific action when the user right-clicks on one of the cells that contains summarized data. The *ICol* and *IRow* parameters indicate the column and row indexes of the cell. The column and row indexes specify the value 0 for the top or leftmost cell in the scrollable data region, not the top left cell of the decision grid. The *ISum* parameter is the index of the current summary in the decision cube, where the first active summary has an index of 0. The *ValueArray* parameter gives the coordinates of the cell in terms of the dimensions of the decision source. If the cell represents a subtotal value, the entry in the *ValueArray* for the dimension over which the subtotal is calculated is a negative number.

Use the *Cells* property to obtain the value of the summary in the cell. Use the *GetDimensionName* method of the decision source to get the name of each dimension represented by an entry in the *ValueArray* parameter. Use the *GetValueAsString* or *GetValueAsVariant* method of the decision source to obtain the label values for each dimension represented in the *ValueArray* parameter.

See also *Cells* property, *DecisionSource* property, *GetValueArray* method, *TDecisionSource.GetDimensionName* method, *TDecisionSource.GetValueAsString* method, *TDecisionSource.GetValueAsVariant* method, *TSmallIntArray* object

OnSelectCell

OnSelectCell occurs when before a cell is selected.

```
type TSelectCellEvent = procedure (Sender: TObject; Col, Row: Longint;
  var CanSelect: Boolean) of object;

property OnSelectCell: TSelectCellEvent;
```

Description Write an *OnSelectCell* event handler to specify whether individual cells in the grid can be selected. Set the *CanSelect* parameter to *False* to block the user from selecting the cell indicated by the *Col* and *Row* parameters. *Col* and *Row* are the indexes of the cell the user is attempting to select, where (0,0) indicates the top left cell of the grid.

See also *GetTabOrderList* method, *OnDecisionExamineCell* event

OnTopLeftChanged

OnTopLeftChanged occurs when the data region of the grid scrolls.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnTopLeftChanged: TNotifyEvent;
```

Description Write an *OnTopLeftChanged* event handler to take specific action when the summarized data in the grid scrolls. Use the *Cells* property or the *CellDrawState* method to obtain the new values of the data cells.

See also *Cells* property, *CellDrawState* method, *CellValueArray* method

Methods

The following table lists the methods contained in *TDecisionGrid*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
CellDrawState	public	TCustomDecisionGrid
CellValueArray	public	TCustomDecisionGrid
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ColWidthsChanged	protected	TCustomDecisionGrid
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomDecisionGrid
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCustomDecisionGrid
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
DrawCell	protected	TCustomDecisionGrid
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetCells	protected	TCustomDecisionGrid

Method	Visibility	Documented in
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
MouseCoord	public	TCustomGrid
MouseDown	protected	TCustomDecisionGrid
MouseMove	protected	TCustomDecisionGrid
MouseUp	protected	TCustomDecisionGrid
NewInstance	public	TObject
Notification	protected	TCustomDecisionGrid
Paint	protected	TCustomDecisionGrid
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
RowHeightsChanged	protected	TCustomDecisionGrid
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectCell	protected	TCustomDecisionGrid
SendToBack	public	TControl
SetBounds	public	TWinControl

Method	Visibility	Documented in
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
TopLeftChanged	protected	TCustomDecisionGrid
Update	public	TWinControl
UpdateControlState	public	TWinControl

CellDrawState

CellDrawState retrieves information about a data cell in the grid.

```
type
  TDecisionDrawStates = (dsGroupStart, dsRowCaption, dsColCaption, dsHeader, dsSum,
    dsRowValue, dsColValue, dsData, dsOpenAfter, dsCloseAfter, dsCloseBefore, dsOpenBefore,
    dsRowIndicator, dsColIndicator, dsRowPlus, dsColPlus, dsNone,dsLeftPivot, dsMiddlePivot,
    dsRightPivot, dsTopPivot);
  TDecisionDrawState = set of TDecisionDrawStates;

function CellDrawState(ACol, ARow: Integer; var Value: string;
  var DrawState: TDecisionDrawState): boolean;
```

Description Call *CellDrawState* to obtain information about a cell in the scrollable data region of the grid. The *ACol* and *ARow* parameters specify the index of the cell, where (0,0) is the upper left cell in the data region, not the upper left cell in the grid. The *Value* parameter returns the value of the summary in the cell, formatted for display. The *DrawState* parameter returns information about the type of information contained in the cell. For more information about the possible values of *DrawState*, see the entry for the *OnDecisionDrawCell* event.

CellDrawState returns *True* if the *Value* and *DrawState* parameters are correctly filled in. *CellDrawState* returns *False* if the cell specified by *ACol* and *ARow* is not in the data region of the grid.

See also *Cells* property, *OnDecisionDrawCell* event

CellValueArray

CellValueArray returns the values of all dimension fields over which a data cell is summarized.

```
function CellValueArray(ACol, ARow: Integer; var ValueArray: TSmallIntArray): boolean;
```

Description Call *CellValueArray* to obtain the coordinates of a data cell in the multidimensional array maintained by the decision cube. The *ACol* and *ARow* parameters represent the indexes of the column and row containing the cell, where (0,0) specifies the upper left data cell, not the top left cell in the grid. The *ValueArray* parameter returns the indexes of the field values for each dimension over which the cell summarizes data. Use the *GetMemberAsString* or *GetMemberAsVariant* method of the decision source to

convert the indexes in the *ValueArray* parameter into field values. *CellValueArray* returns a boolean indicating whether the function was successful.

See also *TDecisionSource.GetMemberAsString* method, *TDecisionSource.GetMemberAsVariant* method, *TSmallIntArray* object

ColWidthsChanged

ColWidthsChanged responds when the column widths change.

```
procedure ColWidthsChanged; override;
```

Description *ColWidthsChanged* is called immediately after the column widths change. The change can result from setting the *ColWidth* property or from resizing the columns with the mouse.

ColWidthsChanged adjusts the scroll bar to accommodate the change. At design time, *ColWidthsChanged* forces all columns to have the same size when the columns are resized using the mouse.

See also *ColWidth* property, *TCustomGrid.ScrollBars* property

Create

Create creates and initializes an instance of *TCustomDecisionGrid*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to create a decision grid at runtime. Decision grids placed on forms at design time are created automatically.

Create initializes the following properties:

- *CaptionColor* to *clActiveCaption* and *CaptionFont.Color* to *clCaptionText*.
- *DataColor* to *clInfoBk* and *DataSumColor* to *clNone*.
- *LabelColor* to *clBtnFace* and *LabelSumColor* to *clTeal*.
- *Color* to *clWindow* and *ParentColor* to *False*.
- *Options* to [*cgGridLines, cgOutliner, cgPivotable*].
- *GridLineColor* to *clBlack* and *GridLineWidth* to 1.
- *ColWidth* to 80 and *RowHeight* to 20.
- *ScrollBars* to *ssBoth*.
- *DefaultDrawing* to *True*.

See also *CaptionColor* property, *CaptionFont* property, *ColWidth* property, *DataColor* property, *DataSumColor* property, *GridLineColor* property, *GridLineWidth* property, *LabelColor* property, *LabelSumColor* property, *Options* property, *RowHeight* property, *TControl.Color* property, *TControl.ParentColor* property, *TCustomGrid.Create* method, *TCustomGrid.DefaultDrawing* property, *TCustomGrid.ScrollBars* property

Destroy

Destroy destroys an instance of *TCustomDecisionGrid*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the decision grid is not already freed, and only then calls *Destroy*. *Destroy* frees all helper objects used by *TCustomDecisionGrid*.

See also *Free* method

DrawCell

DrawCell draws a cell in the grid.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState); override;
```

Description *DrawCell* is called automatically when the cell specified by the *ACol* and *ARow* parameters needs to be painted. *DrawCell* generates an *OnDecisionDrawCell* event, and then draws the cell using the color, font, and value for the cell.

See also *OnDecisionDrawCell* event

GetCells

GetCells returns the value of a cell in the grid, formatted as a string.

```
function Cells(ACol, ARow: Integer): String;
```

Description *GetCells* is the protected property access method for the *Cells* property. Override *GetCells* to change the implementation of the *Cells* property.

See also *Cells* property

MouseDown

MouseDown starts the drag of a dimension to a new location.

```
procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); override;
```

Description *MouseDown* is called automatically when the user clicks the mouse in a cell of the grid. After calling the inherited method, *MouseDown* starts a pivot operation if the cell is a dimension label that can be dragged.

When overriding the *MouseDown* method, be sure to call the inherited method.

See also *MouseMove* method, *MouseUp* method, *TCustomGrid.MouseDown* method

MouseMove

MouseMove computes the drop site of a pivoting dimension.

```
procedure MouseMove(Shift: TShiftState; X, Y: Integer); override;
```

Description *MouseMove* is called automatically when the user moves the mouse. After calling the inherited method, *MouseMove* calculates the new pivot state if a dimension is being dragged to a new location.

When overriding the *MouseMove* method, be sure to call the inherited method.

See also *MouseDown* method, *MouseUp* method, *TCustomGrid.MouseMove* method

MouseUp

MouseUp rearranges the dimensions after a pivot and updates the decision source.

```
procedure MouseUp(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); override;
```

Description *MouseUp* is called automatically when the user releases the mouse. After calling the inherited method, *MouseUp* checks whether a dimension was just dragged to a new location. If a dimension was just moved, *MouseUp* changes the *Dimensions* property appropriately and updates the decision source to reflect the new configuration.

When overriding the *MouseUp* method, be sure to call the inherited method.

See also *Dimensions* property, *MouseDown* method, *MouseMove* method, *TCustomGrid.MouseUp* method

Notification

Notification links the buttons in *TDecisionPivot* to the decision source.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description After calling the inherited method, *Notification* performs the necessary internal adjustments when new elements are added to the decision grid. When overriding the *Notification* method, be sure to call the inherited method.

See also *DecisionSource* property, *TDecisionPivot* object

Paint

Paint renders the image of the decision grid.

```
procedure Paint; override;
```

Description *Paint* is called when the grid receives a Windows paint message. *Paint* draws the grid, calling the *DrawCell* method when it is time to paint the contents of each cell. After all the cells are drawn, *Paint* draws the grid lines.

If *DefaultDrawing* is *False*, *Paint* does not draw the background of the cells or draw the focus rectangle around the selected cell when it has focus.

See also *DrawCell* method, *TCustomGrid.DefaultDrawing* method

RowHeightsChanged

RowHeightsChanged responds when the row heights change.

```
procedure RowHeightsChanged; override;
```

Description *RowHeightsChanged* is called immediately after the row heights change. The change can result from setting the *RowHeight* property or from resizing the rows with the mouse.

RowHeightsChanged adjusts the scroll bar to accommodate the change. At design time, *RowHeightsChanged* forces all rows to have the same size when the rows are resized using the mouse.

See also *RowHeight* property, *TCustomGrid.ScrollBars* property

SelectCell

SelectCell generates an *OnSelectCell* event.

```
function SelectCell(Arow, AColumn:LongInt): Boolean; override;
```

Description Decision grids call *SelectCell* internally before attempting to move selection to a the cell identified by the *ACol* and *ARow* parameters. *ACol* and *ARow* are the column and row indexes for the cell, where the first column has index 0, and the top row has index 0.

See also *OnSelectCell* event

TopLeftChanged

TopLeftChanged generates an *OnTopLeftChanged* event.

```
procedure TopLeftChanged; override;
```

Description Decision grids call *TopLeftChanged* internally, whenever the data region of the grid scrolls.

See also *OnTopLeftChanged* event

TCustomEdit

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomEdit

The *TCustomEdit* object is the abstract class from which all edit-box and memo controls are derived.

Description *TCustomEdit* encapsulates the fundamental behavior common to all components for editing text by introducing methods and properties that provide

- Basic text editing functions such as selecting text, modifying selected text, and case conversions.
- Ability to respond to changes in the contents of the text.
- Access control of the text for making it read-only or introducing a password character to hide the actual value.

Do not create instances of *TCustomEdit*. Use *TCustomEdit* as a base class when declaring control objects that permit the user to enter or modify text. Properties and methods of *TCustomEdit* provide basic behavior that descendent classes inherit as well as behavior that components can override to customize their behavior.

See also *TCustomMaskEdit* object, *TCustomMemo* object, *TEdit* object

Properties

The following table lists the properties of *TCustomEdit*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
AutoSelect	protected	True	TCustomEdit
AutoSize	protected	True	TCustomEdit
BorderStyle	protected	bsSingle	TCustomEdit
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
CharCase	protected	ecNormal	TCustomEdit
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent

Property	Visibility	Default	Documented in
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
HideSelection	protected	True	TCustomEdit
Hint	published		TControl
Left	published		TControl
MaxLength	protected	0	TCustomEdit
Modified	public		TCustomEdit
Name	published		TControl
OEMConvert	protected	False	TCustomEdit
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	False	TControl
ParentWindow	public		TWinControl
PasswordChar	protected	#0	TCustomEdit
ReadOnly	protected	False	TCustomEdit
SelLength	public		TCustomEdit
SelStart	public		TCustomEdit
SelText	public		TCustomEdit
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	public		TControl
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

AutoSelect

AutoSelect determines whether all the text in the edit control is automatically selected when the control gets focus.

property AutoSelect: Boolean;

Description Set *AutoSelect* to select all the text when the edit control gets focus. *AutoSelect* only applies to single-line edit controls.

Use *AutoSelect* when the user is more likely to replace the text in the edit control than to append to it.

See also *SelectAll* method, *SelLength* property, *SelStart* property, *SelText* property, *SetSelTextBuf* method

AutoSize

AutoSize determines whether the height of the edit control automatically resizes to accommodate the text.

property AutoSize: Boolean;

Description Use *AutoSize* to make the edit control adjust its size automatically so the client area accommodates the height of the text. When *AutoSize* is *False*, the edit control has a fixed height. When *AutoSize* is *True*, the size of the control is readjusted whenever a change occurs that could affect the height of the control, such as a change to the font or border style.

Note *AutoSize* only adjusts the height of the edit control Even with *AutoSize* set to *True*, text in the edit control may appear truncated if it extends beyond the width of the edit control.

See also *BorderStyle* property, *ClientHeight* property, *ControlStyle* property, *Height* property

BorderStyle

BorderStyle determines whether the edit control has a single line border around the client area.

property BorderStyle: TBorderStyle;

Description Use *BorderStyle* to affect the sharpness with which the client area of the edit control stands out. *BorderStyle* can have a value of either *bsSingle* or *bsNone*. If *BorderSyle* is *bsSingle*, the edit control has a single-line border around the client area. If *BorderStyle* is *bsNone*, there will be no border.

Note *BorderStyle* interacts with the *Ctl3D* property to create different visual effects. Try combining different combinations of the two properties to get the desired look.

See also *TWinControl.Ctl3D* property

CharCase

CharCase determines the case of the text within the edit control.

property CharCase: TEditCharCase;

Description Use *CharCase* to force the contents of the edit control to assume a particular case. The possible values of *CharCase* are as follows:

Value	Meaning
ecLowerCase	The text is converted to lowercase.
ecNormal	The text appears in mixed case. It is not forced into any case.
ecUpperCase	The text is converted to uppercase.

When *CharCase* is set to *ecLowerCase* or *ecUpperCase*, the case of characters is converted as the user types them into the edit control. Changing the *CharCase* property to *ecLowerCase* or *ecUpperCase* changes the actual contents of the text, not just the appearance. Any case information is lost and can't be recaptured by changing *CharCase* to *ecNormal*.

See also *Text* property

HideSelection

HideSelection determines whether the visual indication of the selected text remains when focus shifts to another control.

property HideSelection: Boolean;

Description Set *HideSelection* to *False* to provide visual feedback of the selected portion of the text even when the edit control does not have focus. Set *HideSelection* to *True* to show the selection only when the edit control has focus. *HideSelection* does not affect the actual value of the selection, only the visual indication. Always setting *HideSelection* to *False* can make forms with many edit controls look too busy.

See also *SelLength* property, *SelStart* property, *SelText* property, *SetSelTextBuf* method

MaxLength

MaxLength specifies the maximum number of characters the user can enter into the edit control.

property MaxLength: Integer;

Description Use *MaxLength* to limit the number of characters that can be entered into the edit control. A value of 0 indicates that there is no application-defined limit on the length. *MaxLength* only applies to single-line edit controls.

Use *MaxLength* to limit the length of the text in an edit control if that text will be copied into a fixed-length buffer.

Note Setting *MaxLength* will not truncate the existing text, it merely prevents the user from adding more text after reaching the limit of *MaxLength* characters.

See also *Text* property

Modified

Modified indicates whether the user edited the text of the edit control.

```
property Modified: Boolean;
```

Description Use *Modified* to determine whether the *Text* property of the edit control changed. If an application directly alters the *Text* property of an edit control, it should set the *Modified* property to *True*.

See also *Change* method, *OnChange* event, *SetTextBuf* method, *Text* property

OEMConvert

OEMConvert determines whether characters typed in the edit control are converted from ANSI to OEM and then back to ANSI.

```
property OEMConvert: Boolean;
```

Description To ensure that any characters typed in the edit control can be unambiguously converted to the OEM character set, set *OEMconvert* to *True*. This causes all characters to be converted from ANSI to OEM and then back to ANSI as they are typed. Set *OEMconvert* to *False* to avoid the overhead of this conversion when it does not matter whether the text can be unambiguously mapped to an OEM string.

OEMConvert is most useful for edit controls used to enter file names when the application does not use Unicode file names.

See also *Text* property

PasswordChar

PasswordChar indicates the character, if any, to display in place of the actual characters typed in the control.

```
property PasswordChar: Char;
```

Description Use the *PasswordChar* property to create an edit control that displays a special character in place of any entered text. If *PasswordChar* is set to the null character (ANSI character zero), the edit control displays its text normally. If *PasswordChar* is any other character, the edit control displays *PasswordChar* in place of each character typed. *PasswordChar* affects the appearance of the edit control only. The value of the *Text* property reflects the actual characters that are typed.

See also *Text* property

ReadOnly

ReadOnly determines whether the user can change the text of the edit control.

property ReadOnly: Boolean;

Description To restrict the edit control to display only, set the *ReadOnly* property to *True*. Set *ReadOnly* to *False* to allow the contents of the edit control to be edited.

Setting *ReadOnly* to *True* ensures that the text is not altered, while still allowing the user to select text. The selected text can then be manipulated by the application, or copied to the Clipboard.

See also *CopyToClipboard* method, *GetSelTextBuf* method

SelLength

SelLength is the number of characters that are selected.

property SelLength: Integer;

Description Read *SelLength* to determine the length, in characters, of the selected text. Set *SelLength* to change the selection to consist of the first *SelLength* characters starting at *SelStart*.

Note Setting *SelLength* to a value greater than the number of characters from *SelStart* to the end of the text results in the selection of all characters from *SelStart* to the end of the text. Reading *SelLength* immediately after setting it to a value greater than the number of available characters returns the number of characters actually selected, not the value that was just set.

See also *SelStart* property, *SelText* property

SelStart

SelStart is the position of the first selected character in the text.

property SelStart: Integer;

Description Read *SelStart* to determine the position of the first selected character, where 0 indicates the first character. If there is no selected text, *SelStart* indicates the position of the cursor. Set *SelStart* to remove the current selection and position the cursor just before the indicated character.

To select a particular range of the text, first set *SelStart* to position the cursor, and then set *SelLength* to extend the selection.

See also *SelLength* property, *SelText* property

SelText

SelText is the selected portion of the edit control's text.

property SelText: string;

Description Read *SelText* to determine the value of the selected text. Set *SelText* to replace the selected text with a new string. If there is no selection, but the edit control has focus, set *SelText* to insert a new string into the text at the cursor.

See also *ClearSelection* method, *GetSelTextBuf* method, *SelLength* property, *SelStart* property, *SetSelTextBuf* method, *Text* property

Events

The following table lists the events of *TCustomEdit*:

Event	Visibility	Documented in
OnChange	protected	TCustomEdit

OnChange

OnChange occurs whenever the text for the edit control may have changed.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property OnChange: TNotifyEvent;
```

Description Write an *OnChange* event handler to take specific action whenever the text for the edit control may have changed. Use the *Modified* property to see if a change actually occurred. The *Text* property of the edit control will already be updated to reflect any changes. This event provides the first opportunity to respond to modifications that the user types into the edit control.

See also *Change* method, *Text* property

Methods

The following table lists the methods of *TCustomEdit*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
Change	protected	TCustomEdit
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject

Method	Visibility	Documented in
CleanupInstance	public	TObject
Clear	public	TCustomEdit
ClearSelection	public	TCustomEdit
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
CopyToClipboard	public	TCustomEdit
Create	public	TCustomEdit
CreateParams	protected	TCustomEdit
CreateParented	public	TWinControl
CreateWnd	protected	TCustomEdit
CutToClipboard	public	TCustomEdit
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DestroyWnd	protected	TCustomEdit
DisableAlign	public	TWinControl
Dispatch	public	TObject
DoSetMaxLength	protected	TCustomEdit
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetSelLength	protected	TCustomEdit
GetSelStart	protected	TCustomEdit
GetSelText	protected	TCustomEdit
GetSelTextBuf	public	TCustomEdit
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl

Method	Visibility	Documented in
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
PasteFromClipboard	public	TCustomEdit
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectAll	public	TCustomEdit
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetSelLength	protected	TCustomEdit
SetSelStart	protected	TCustomEdit
SetSelTextBuf	public	TCustomEdit
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Change

Change responds to changes in the *Text* property by calling the *OnChange* event handler if it is assigned.

procedure Change; **dynamic**;

Description Applications do not call the *Change* method directly. It is triggered automatically by Windows events. *Change* is exposed as a protected method so that derived classes can make additional responses to changes in the *Text* property beyond invoking the *OnChange* event handler. Derived classes that override the *Change* method should be sure to call the inherited method.

See also *Handle* property, *OnChange* event, *Text* property

Clear

Clear deletes all text from the edit control.

procedure Clear; **virtual**;

Description Use *Clear* to replace the *Text* property with an empty string. *Clear* removes all text, not just the selected text. To remove just the selected text, use *ClearSelection*.

See also *ClearSelection* method, *Text* property

ClearSelection

ClearSelection removes the selected text from the edit control.

procedure ClearSelection;

Description Use *ClearSelection* to delete the selected text from the edit control. If no text is selected, *ClearSelection* does nothing. If all of the text is selected, *ClearSelection* clears all text, like the *Clear* method.

Note Calling *ClearSelection* does the same thing as sending the edit control a WM_CLEAR message.

See also *Clear* method, *CutToClipboard* method, *SelText* property, *SetSelTextBuf* method

CopyToClipboard

CopyToClipboard copies the selected text in the edit control to the Clipboard in CF_TEXT format.

procedure CopyToClipboard;

Description Use *CopyToClipboard* to replace the contents of the Clipboard with the selected text. *CopyToClipboard* does not clear the Clipboard if no text is selected. If no text is selected, *CopyToClipboard* does nothing.

Note Calling *CopyToClipboard* does the same thing as sending the edit control a WM_COPY message.

See also *CutToClipboard* method, *GetSelTextBuf* method, *PasteFromClipboard* method, *SelText* property

Create

Create creates an instance of *TCustomEdit*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to create and initialize an instance of *TCustomEdit*.

After calling the inherited constructor, *TCustomEdit* initializes

- *AutoSelect* to *True*.
- *AutoSize* to *True*.
- *BorderStyle* to *bsSingle*.
- *ControlStyle* to [*csClickEvents, csDoubleClicks, csFixedHeight, csSetCaption*].
- *Height* and *Width* to 25 and 121.
- *HideSelection* to True.
- *ParentColor* to False.
- *TabStop* to True.

Derived objects that override the *Create* method should call the inherited *Create* before performing any additional initialization.

See also *AutoSelect* property, *AutoSize* property, *BorderStyle* property, *ControlStyle* property, *Height* property, *HideSelection* property, *ParentColor* property, *TabStop* property, *TWinControl.Create* method, *Width* property

CreateParams

The *CreateParams* method initializes a window-creation parameter record passed in the *Params* parameter.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;

procedure CreateParams(var Params: TCreateParams); override;
```

Description The *CreateWnd* method calls *CreateParams* to initialize the parameters it passes to *CreateWindowHandle*. *TCustomEdit* overrides the inherited method to specify a scrollable single-line edit control that reflects the current values of the *BorderStyle, HideSelections, PasswordChar,* and *ReadOnly* properties.

See also *BorderStyle* property, *CreateWindowHandle* method, *CreateWnd* method, *HideSelections* property, *PasswordChar* property, *ReadOnly* property

CreateWnd

The *CreateWnd* method creates a Windows control corresponding to the edit control.

```
procedure CreateWnd; override;
```

Description *CreateWnd is called when the edit control is created, and whenever a change to the edit control requires the window to be recreated.* After calling the inherited method, *CreateWnd* sends the newly created window messages to enforce the *MaxLength* and *PasswordChar* properties.

See also *CreateParams* method, *CreateWindowHandle* method, *DestroyWnd* method, *MaxLength* property, *PasswordChar* property, *TWinControl.CreateWnd* method

CutToClipboard

CutToClipboard copies the selected text in the edit control to the Clipboard in CF_TEXT format and then deletes the selection.

```
procedure CutToClipboard;
```

Description Use *CutToClipboard* to replace the contents of the Clipboard with the selected text while clearing the selected text. If no text is selected, *CutToClipboard* does nothing.

See also *ClearSelection* method, *CopyToClipboard* method, *GetSelTextBuf* method, *PasteFromClipboard* method, *SelText* property

DestroyWnd

DestroyWnd destroys the Windows control that corresponds to the edit control.

```
procedure DestroyWnd; override;
```

Description Before the window is destroyed, *DestroyWnd* saves a copy of any window properties that would be needed to recreate the window after it is destroyed.

See also *CreateWnd* method

DoSetMaxLength

DoSetMaxLength sets the *MaxLength* property.

```
procedure DoSetMaxLength(Value: Integer); virtual;
```

Description *DoSetMaxLength* is the property access function for the *MaxLength* property. It is exposed as a protected method so that descendants of *TCustomEdit* can customize the implementation of the *MaxLength* property.

DoSetMaxLength sends an EM_LIMITTEXT message to the window of the edit control.

See also *MaxLength* property

GetSelLength

GetSelLength returns the *SelLength* property.

```
function GetSelLength: Integer; virtual;
```

Description *GetSelLength* is the property access function for the *SelLength* property. It is exposed as a protected method so that descendants of *TCustomEdit* can customize the implementation of the *SelLength* property.

GetSelLength sends an EM_GETSEL message to the window of the edit control and returns the difference between the starting position and the ending position.

See also *SelLength* property, *SetSelLength* method

GetSelStart

GetSelStart returns the *SelStart* property.

```
function GetSelStart: Integer; virtual;
```

Description *GetSelStart* is the property access method for the *SelStart* property. It is exposed as a protected method so that descendants of *TCustomEdit* can customize the implementation of the *SelStart* property.

GetSelLength sends an EM_GETSEL message to the window of the edit control and returns the starting position.

See also *SelStart* property, *SetSelStart* method

GetSelText

GetSelText returns the *SelText* property.

```
function GetSelText: string; virtual;
```

Description *GetSelText* is the property access method for the *SelText* property. It is exposed as a protected method so that descendants of *TCustomEdit* can customize the implementation of the *SelText* property.

See also *SelText* property

GetSelTextBuf

GetSelTextBuf copies the selected text into a buffer and returns the number of characters copied.

```
function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer; virtual;
```

Description Use *GetSelTextBuf* to copy the selected text into a character buffer. If there is no selection, the buffer receives an empty string. If the selection contains more than (*BufSize* – 1) characters, only the first (*BufSize* – 1) characters are copied. *GetSelTextBuf* returns the number of characters that were actually copied into the buffer.

See also *CopyToClipboard* method, *GetTextBuf* method, *SelLength* property, *SelText* property, *SetSelTextBuf* method

PasteFromClipboard

PasteFromClipboard copies the contents of the Clipboard into edit control, replacing the current selection.

```
procedure PasteFromClipboard;
```

Description Use *PasteFromClipboard* to replace the selected text with the contents of the Clipboard, or, if no text is selected, to insert the contents of the Clipboard at the cursor. If the Clipboard is empty, or if it does not contain anything in CF_TEXT format, *PasteFromClipboard* does nothing.

Note Calling *PasteFromClipboard* does the same thing as sending the edit control a WM_PASTE message.

See also *CutToClipboard* method, *PasteFromClipboard* method, *SelStart* property, *SelText* property, *SetSelTextBuf* method

SelectAll

SelectAll selects all text in the edit control.

```
procedure SelectAll;
```

Description Use *SelectAll* to select all text in the edit control. To select only part of the text, use the *SelStart* and *SelLength* properties.

See also *SelLength* property, *SelStart* property, *SelText* property, *Text* property

SetSelLength

SetSelLength sets the *SelLength* property.

```
procedure SetSelLength(Value: Integer); virtual;
```

Description *SetSelLength* is the property access method for the *SelLength* property. It is exposed as a protected method so that descendants of *TCustomEdit* can customize the implementation of the *SelLength* property.

SetSelLength sends an EM_SETSEL message to the window of the edit control, specifying a selection that begins at *SelStart* and ends at *SelStart* + *Value*.

See also *GetSelLength* method, *SelLength* property, *SelStart* property

SetSelStart

SetSelStart returns the *SelStart* property.

```
procedure SetSelStart(Value: Integer); virtual;
```

Description *SetSelStart* is the property access method for the *SelStart* property. It is exposed as a protected method so that descendants of *TCustomEdit* can customize the implementation of the *SelStart* property.

SetSelLength sends an EM_SETSEL message to the window of the edit control to position the cursor on the position indicated by *Value*.

See also *GetSelStart* method, *SelStart* property

SetSelTextBuf

SetSelTextBuf replaces the selected text with a null-terminated string.

```
procedure SetSelTextBuf(Buffer: PChar);
```

Description Use *SetSelTextBuf* to replace the current selection by the contents of the null-terminated string pointed to by *Buffer*. If no text is selected, the contents of *Buffer* are inserted at the cursor. *SetSelTextBuf* does the same thing as setting the *SelText* property, except that it takes a *PChar* rather than a *string*.

See also *GetSelTextBuf* method, *PasteFromClipboard* method, *SelText* property, *SetTextBuf* method

TCustomForm

Unit: Forms

TObject • TPersistent • TComponent • TControl • TWinControl • TScrollingWinControl • TCustomForm

The *TCustomForm* component is the abstract base class from which to derive an application form.

Description Derive from *TCustomForm* to create a custom window. The *TCustomForm* derivative can contain other objects, such as *TButton*, *TCheckBox*, and *TComboBox* objects.

See also *TPropertyPage* object, *TActiveForm* object, *TForm* object

Properties

The following table lists the properties of *TCustomForm:*

Property	Visibility	Default	Documented in
Active	public, read-only		TCustomForm
ActiveControl	public		TCustomForm
ActiveMDIChild	protected, read-only		TCustomForm
ActiveOleControl	public		TCustomForm
Align	public	alNone	TControl
AutoScroll	protected	True	TScrollingWinControl
BorderIcons	protected	[biSystemMenu, biMinimize, biMaximize]	TCustomForm
BorderStyle	protected	bsSizeable	TCustomForm
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomForm
Caption	public		TControl
ClientHandle	protected, read-only		TCustomForm
ClientHeight	protected		TCustomForm
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TCustomForm
ClientWidth	protected		TCustomForm
Color	public	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	protected	True	TWinControl
Cursor	published	crDefault	TControl
Designer	public		TCustomForm
DesignInfo	public		TComponent
DropTarget	public		TCustomForm
Enabled	public	True	TControl
Font	public		TControl
FormStyle	protected	fsNormal	TCustomForm
Handle	public, read-only		TWinControl
Height	protected		TControl
HelpContext	published	0	TWinControl
HelpFile	public		TCustomForm

Property	Visibility	Default	Documented in
Hint	published		TControl
HorzScrollBar	protected		TScrollingWinControl
Icon	protected		TCustomForm
KeyPreview	public	False	TCustomForm
Left	published		TControl
MDIChildCount	protected, read-only		TCustomForm
MDIChildren	protected, read-only		TCustomForm
Menu	public		TCustomForm
ModalResult	public		TCustomForm
Name	published		TControl
ObjectMenuItem	protected		TCustomForm
OleFormObject	public		TCustomForm
Owner	public, read-only		TComponent
Parent	public		TCustomForm
ParentFont	protected	False	TControl
ParentWindow	public		TWinControl
PixelsPerInch	protected		TCustomForm
PopupMenu	protected		TControl
Position	protected	poDesigned	TCustomForm
PrintScale	protected	poProportional	TCustomForm
Scaled	protected	True	TCustomForm
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
TileMode	protected	tbHorizontal	TCustomForm
Top	published		TControl
VCLComObject	public		TComponent
VertScrollBar	protected		TScrollingWinControl
Visible	protected	False	TCustomForm
Width	protected		TControl
WindowMenu	protected		TCustomForm
WindowProc	public		TControl
WindowState	public	wsNormal	TCustomForm

Active

Active specifies whether the form has focus.

property Active: Boolean;

Description Use *Active* to determine whether the form has focus.

If *Active* is *True*, the form has focus; if *Active* is *False*, the form does not have focus.

See also *SetFocus* method, *OnActivate* event, *OnDeactivate* event

ActiveControl

ActiveControl is the control that has focus on the form.

```
property ActiveControl: TWinControl;
```

Description Use *ActiveControl* to get or set the control that has focus on the form.

If the form does not have focus, *ActiveControl* is the control on the form that will receive focus when the form receives focus.

Note When focus shifts to another control, *ActiveControl* is updated before the *OnExit* event occurs.

Only one control can have focus at a given time in an application.

See also *ActiveOLEControl* property, *OnExit* event, *TWinControl* component, *OnEnter* event

ActiveMDIChild

ActiveMDIChild is the MDI child that has focus.

```
property ActiveMDIChild: TForm;
```

Description Use *ActiveMDIChild* to get the MDI child that has focus.

If the form is not an MDI parent (for example, if the *FormStyle* property of the form is not *fsMDIForm*), *ActiveMDIChild* returns **nil**.

See also *FormStyle* property, *MDIChildCount* property

C

ActiveOleControl

ActiveOLEControl is an OLE control on the form that reacts to changes in focus.

```
property ActiveOLEControl: TWinControl;
```

Description Use *ActiveOLEControl* to get or set an OLE control on the form that reacts to changes in focus.

If *ActiveOLEControl* is not **nil**, the user interface for the specified OLE control is deactivated when it is not focused. Also, when the form receives or loses focus, the document window for the specified OLE control is activated or deactivated respectively.

See also *ActiveControl* property, *TWinControl* component

BorderIcons

BorderIcons is a set of values that specify which icons appear on the title bar of the form.

property BorderIcons: TBorderIcons;

Description Use *BorderIcons* to get or set the icons that appear on the title bar of the form. *BorderIcons* can consist of any of the following *TBorderIcons* values:

Value	Meaning
biSystemMenu	The form has a Control menu (also known as a System menu)
biMinimize	The form has a Minimize button
biMaximize	The form has a Maximize button
biHelp	If BorderStyle is bsDialog or biMinimize and biMaximize are excluded, a question mark appears in the form's title bar and when clicked, the cursor changes to crHelp; otherwise, no question mark appears.

Certain combinations of the *BorderIcons* and *BorderStyle* properties are mutually exclusive. For example, *BorderIcons biMax, biMin* with *BorderStyle* of *bsDialog* are mutually exclusive.

See also *BorderStyle* property

BorderStyle

BorderStyle is a set of values that specify the appearance and behavior of the form border.

property BorderStyle: TFormBorderStyle;

Description Use *BorderStyle* to get or set the appearance and behavior of the form border. *BorderStyle* can consist of any of the following *TFormBorderStyle* values:

Value	Meaning
bsDialog	Not resizable; standard dialog box border
bsSingle	Not resizable; single-line border
bsNone	Not resizable; no visible border line
bsSizeable	Standard resizable border
bsToolWindow	like bsSingle but with a smaller caption
bsSizeToolWin	like bsSizeable with a smaller caption

Note Changing the border style of an MDI child form to bsDialog or bsNone has no effect.

See also *BorderIcons* property

Canvas

Canvas is the drawing area of the application.

property Canvas: TCanvas;

Description Use *TCanvas* to get the drawing area of the application. *Canvas* is often used in the *OnPaint* event handler.

See also *TCanvas* component

ClientHandle

ClientHandle is the handle (HWND) for the form's internal MDI client window.

property ClientHandle: HWND;

Description Use *ClientHandle* to get the handle for the form's internal MDI client window.

ClientHandle is meaningful only if the form is an MDI parent (for example, if the form's *FormStyle* property is set to *fsMDIForm*).

See also *FormStyle* property

ClientHeight

ClientHeight is the height (in pixels) of the form's client area.

property ClientHeight: Integer;

Description Use *ClientHeight* to get or set the height (in pixels) of the form's client area. The client area is the usable area inside the form's border.

See also *ClientRect* property, *ClientWidth* property

C

ClientRect

ClientRect is the dimensions of the form's client area.

property ClientRect: TRect;

Description Use *ClientHeight* to get or set the form's client area. The client area is the usable area inside the form's border.

ClientRect is equivalent to Rect(0,0,ClientWidth, ClientHeight).

See also *ClientHeight* property, *ClientWidth* property, *Rect* function, *TRect* type

ClientWidth

ClientWidth is an integer that represents the width (in pixels) of the form's client area.

property ClientWidth: Integer;

Description Use *ClientWidth* to get or set the width (in pixels) of the form's client area. The client area is the usable area inside the form's border.

See also *ClientHeight* property, *ClientRect* property

Designer

Designer is the designer for the form.

property Designer: TDesigner;

Description The Designer property is used internally by the form designer in Delphi. You should never assign a value to this property. This is done for you by the form designer. You should only have to use this property if you are designing some classes for use in the form designer such as a property editor or a component editor.

See also *TDesigner* component

DropTarget

DropTarget is a Boolean value that specifies whether the form is the target of a drag-and-drop operation.

property DropTarget: Boolean;

Description Use *DropTarget* to get or set whether the form is the target of a drag-and-drop operation.

See also *BeginDrag* method, *DragMode* method, *EndDrag* method, *OnDragOver* event

FormStyle

FormStyle is a set of values that represent the form's style.

property FormStyle: TFormStyle;

Description Use *FormStyle* to get or set the form's style. *FormStyle* can consist of any combination of the following *TFormStyle* values:

Value	Meaning
fsNormal	The form is neither an MDI parent window nor an MDI child window.
fsMDIChild	The form is an MDI child window.
fsMDIForm	The form is an MDI parent window.
fsStayOnTop	This form remains on top of the desktop and of other forms in the project, except any others that also have FormStyle set to fsStayOnTop.

Note It is not advisable to change *FormStyle* at runtime.

If the form is the main form of an MDI application, its *FormStyle* property must be set to *fsMDIForm*.

See also *ArrangeIcons* method, *Tile* method, *Cascade* method

HelpFile

HelpFile specifies the name of the file the application uses to display Help.

```
property HelpFile: string;
```

Description Use *HelpFile* for forms that have a help file to use with the Windows Help system. Windows displays the file specified by the *HelpFile* property.

See also *HelpContext* property, *OnHelp* event, *TApplication.HelpFile* property

KeyPreview

KeyPreview is a Boolean value that specifies whether keyboard events should occur on the form before they occur on the active control.

```
property KeyPreview: Boolean;
```

Description If *KeyPreview* is *True*, keyboard events occur on the form before they occur on the active control. (The active control is specified by the *ActiveControl* property.)

Navigation keys (*Tab, BackTab,* the arrow keys, and so on) are unaffected by *KeyPreview* because they do not fire keyboard events.

If *KeyPreview* is *False*, keyboard events occur on the active control.

KeyPreview is *False* by default.

See also *ActiveControl* property, *OnKeyDown* event, *OnKeyUp* event, *OnKeyPress* event

C

MDIChildCount

MDIChildCount is the number of open MDI child forms.

```
property MDIChildCount: Integer;
```

Description Use *MDIChildCount* to get the number of open MDI child forms.

MDIChildCount is meaningful only if the form is an MDI parent (for example, if the form's *FormStyle* property is set to fsMDIForm).

See also *MDIChildren* property, *FormStyle* property

MDIChildren

MDIChildren is an array of MDI child forms.

```
property MDIChildren[I: Integer]: TForm;
```

Description Use *MDIChildren* to access a child form.

I is the index of the child form to access. The index of the most recently created MDI child is always 0, and the index of the first-created MDI child is always *MDIChildCount* - 1 (the *MDIChildren* array is reverse-order of MDI child creation).

MDIChildren is meaningful only if the form is an MDI parent (for example, if the form's *FormStyle* property is set to fsMDIForm).

See also *MDIChildCount* property

Menu

Menu is the form's main menu.

```
property Menu: TMainMenu;
```

Description Use *Menu* to get or set the form's main menu.

Menu is set to the first *TMainMenu* component added to the form at design-time.

See also *TMainMenu* component

ModalResult

ModalResult is a value that, when non-zero, closes the form when it is displayed modally.

```
property ModalResult: TModalResult;
```

Description Use *ModalResult* to close the form when it is displayed modally.

By default, *ModalResult* is 0. Set *ModalResult* to any nonzero value to close the form. The value assigned to *ModalResult* becomes the return value of the *ShowModal* function call used to display the form. This property is modified automatically by some components such as the *TButton* object. See *TButton.ModalResult* for details.

ModalResult can be one of the following *TModalResult* values:

Constant	Value	Constant	Value
mrNone	0	*mrIgnore*	idIgnore
mrOk	idOK	*mrYes*	idYes
mrCancel	idCancel	*mrNo*	idNo
mrAbort	idAbort	*mrAll*	mrNo + 1
mrRetry	idRetry		

See also *ShowModal* function, *TButton.ModalResult* property

ObjectMenuItem

ObjectMenuItem is an OLE object menu item that reacts to selections of OLE objects.

```
property ObjectMenuItem: TMenuItem;
```

Description Use *ObjectMenuItem* to get or set a menu item that becomes enabled or disabled when an OLE object on the form is selected or unselected respectively.

The OLE object menu item can be used for standard OLE commands, such as Activate or Convert. (The implementations of these commands are provided by the OLE server.)

See also *OleFormObject* property

OleFormObject

OleFormObject indicates that the form is a frame for an in-place OLE object.

```
property OleFormObject: TOleFormObject;
```

Description *OleFormObject* is used to communicate resize and destroy notifications when a form is used as a frame for an in-place OLE object.

See also *ObjectMenuItem* property

PixelsPerInch

PixelsPerInch is a value that represents the proportion of the font on the system on which the form was designed.

```
property PixelsPerInch: Integer;
```

Description Use *PixelsPerInch* at runtime to change how the form is scaled to the current screen resolution. If *PixelsPerInch* is changed from its default value, the form will not have the same proportion on every screen resolution.

At design-time, *PixelsPerInch* is set automatically when the form is saved. Changes to *PixelsPerInch* in the Object Inspector are seen at runtime only.

The form's *Scaled* property must be *True* for *PixelsPerInch* to have any effect.

See also *Scaled* property

Position

Position is a value that represents the size and placement of the form.

```
property Position: TPosition;
```

Description Use *Position* to get or set the size and placement of the form. *Position* can have one of the following *TPosition* values:

Value	Meaning
poDesigned	The form appears positioned on the screen and with the same height and width as it had at design time.
poDefault	The form appears in a position on the screen and with a height and width determined by Windows. Each time you run the application, the form moves slightly down and to the right. The right side of the form is always near the far right side of the screen, and the bottom of the form is always near the bottom of the screen, regardless of the screen's resolution.
poDefaultPosOnly	The form displays with the size you created it at design time, but Windows chooses its position on the screen. Each time you run the application, the form moves slightly down and to the right. When the form can no longer move down and to the right and keep the same size while remaining entirely visible on the screen, the form displays at the top-left corner of the screen.
poDefaultSizeOnly	The form appears in the position you left it at design time, but Windows chooses its size. The right side of the form is always near the far right side of the screen, and the bottom of the form is always near the bottom of the screen, regardless of the screen's resolution.
poScreenCenter	The form remains the size you left it at design time, but is positioned in the center of the screen.

See also *OnResize* event

PrintScale

PrintScale is a value that represents the proportions of a printed form.

```
property PrintScale: TPrintScale;
```

Description Use *PrintScale* to get or set the proportions of the printed form. *PrintScale* can have one of the following *TPrintScale* values:

Value	Meaning
poNone	No special scaling occurs; therefore, the printed form and how the form appears onscreen may appear squished or stretched.
poProportional	The form is printed so that the printed image is approximately the same visible size as on the screen (WYSIWYG). The form image is scaled so that the printed image is approximately the same visible size as on the screen.
poPrintToFit	The form is printed using the same screen proportions, but in a size that just fits the printed page.

See also *Print* method

Scaled

Scaled is a Boolean value that specifies whether the form is sized according to the value of the *PixelsPerInch* property.

property Scaled: Boolean;

Description If *Scaled* is *True*, and the value of *PixelsPerInch* differs from the current system settings, the form is scaled to a new size. If *Scaled* is *False*, no scaling occurs.

See also *PixelsPerInch* property

TileMode

TileMode is a value that represents how an MDI child form is arranged when the *Tile* method is called.

property TileMode: TTileMode;

Description *TileMode* can have one of the following *TTileMode* values:

Value	Meaning
tbHorizontal	Each form stretches across the width of the parent form
tbVertical	Each form stretches along the height of the parent form

TileMode is meaningful only if the form is an MDI child (for example, if the form's *FormStyle* property is set to fsMDIChild).

See also *FormStyle* property, *Tile* method

Visible

Visible is a Boolean value that indicates whether the form is visible.

property Visible: Boolean;

Description Use *Visible* to get or set whether the form is visible.

If *Visible* is *True*, the form is visible -- unless it is completely obscured by other forms. If *Visible* is *False*, the form is not visible.

You can set *Visible* to *True* and bring the form to the surface of other forms by calling the *Show* or *ShowModal* method.

See also *BringToFront* method, *Show* method, *ShowModal* method

WindowMenu

WindowMenu is the Window menu for an MDI parent form.

property WindowMenu: TMenuItem;

Description Use *WindowMenu* to get or set the Window menu for an MDI parent form. The Window menu is a standard menu in MDI applications. It contains commands that let the user manage the windows in the application. Menu items usually include Cascade, Arrange Icons, Tile, and so on.

The Window menu also lists (at the bottom) the child windows that are currently open in the application. When the user selects one of these windows from the menu, the window becomes the active window in the application.

Although this menu is commonly called the Window menu, it can have any name that corresponds to an existing menu item on the menu bar. The names of open child forms are merged onto the menu automatically at runtime.

WindowMenu is meaningful only if the form is an MDI parent (for example, if the form's *FormStyle* property is set to *fsMDIForm*).

See also *FormStyle* property

WindowState

WindowState is a value that represents how the form appears on the screen.

property WindowState: TWindowState;

Description Use *WindowMenu* to get or set whether the form appears on the screen in its minimized, maximized, or normal state.

WindowState can have one of the following *TWindowState* values:

Value	Meaning
wsNormal	The form appears in its normal state (for example, its non-minimized, non-maximized state).
wsMinimized	The form appears in its minimized state.
wsMaximized	The form appears in its maximized state.

See also *Icon* property, *BorderStyle* property

Events

The following table lists the events of *TCustomForm:*

Event	Visibility	Documented in
OnActivate	protected	TCustomForm
OnClick	protected	TControl
OnClose	protected	TCustomForm
OnCloseQuery	protected	TCustomForm
OnCreate	protected	TCustomForm
OnDblClick	protected	TControl

Event	Visibility	Documented in
OnDeactivate	protected	TCustomForm
OnDestroy	protected	TCustomForm
OnDragDrop	protected	TControl
OnDragOver	protected	TControl
OnHelp	protected	TCustomForm
OnHide	protected	TCustomForm
OnKeyDown	protected	TWinControl
OnKeyPress	protected	TWinControl
OnKeyUp	protected	TWinControl
OnMouseDown	protected	TControl
OnMouseMove	protected	TControl
OnMouseUp	protected	TControl
OnPaint	protected	TCustomForm
OnResize	protected	TCustomForm
OnShow	protected	TCustomForm

OnActivate

The *OnActivate* event for a form occurs when the form becomes active.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnActivate: TNotifyEvent;
```

Description Use *OnActivate* to perform special processing when the form receives focus. A form becomes active when focus is transferred to it (when the user clicks on the form, for example).

For MDI child windows (forms with *FormStyle* property values of *fsMDIChild*), *OnActivate* occurs only when focus is shifted from one child to another. If focus is shifted from a non-MDI child window to an MDI child, the *OnActivate* event occurs for the MDI parent form.

Note The *OnActivate* event of the application (*TApplication*), not the form, occurs when Windows switches control from another application to your application.

See also *FormStyle* property, *OnDeactivate* event

OnClose

OnClose occurs when the form closes.

```
type TCloseEvent = procedure(Sender: TObject; var Action: TCloseAction) of object;

property OnClose: TCloseEvent;
```

Description Use *OnClose* to perform special processing when the form closes. The *OnClose* event specifies which event handler to call when a form is about to

close. The handler specified by *OnClose* might, for example, test to make sure all fields in a data-entry form have valid contents before allowing the form to close.

A form is closed by the *Close* method or when the user chooses *Close* from the form's system menu.

The *TCloseEvent* type points to a method that handles the closing of a form. The value of the *Action* parameter determines if the form actually closes. These are the possible values of *Action*:

Value	Meaning
caNone	The form is not allowed to close, so nothing happens.
caHide	The form is not closed, but just hidden. Your application can still access a hidden form.
caFree	The form is closed and all allocated memory for the form is freed.
caMinimize	The form is minimized, rather than closed. This is the default action for MDI child forms.

If a form is an MDI child form, and its *BorderIcons* property is *biMinimize*, then the default *Action* is *caMinimize*. If a MDI child form does not have these settings, the default *Action* is *caNone*, meaning that nothing happens when the user attempts to close the form.

If a form is an SDI child form, *Action* defaults to *caHide*.

To close and form and free it in an *OnClose* event, set *Action* to *caFree*.

Example This example displays a message dialog box when the user attempts to close the form. If the user clicks the Yes button, the form closes; otherwise, the form remains open.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if MessageDlg('Close application ?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
    Action := caFree
  else
    Action := caNone;
end;
```

See also *OnOpen* event, *OnCloseQuery* event, *OnDestroy* event

OnCloseQuery

The *OnCloseQuery* event occurs when an action to close the form takes place—that is, when the *Close* method is called or when the user chooses *Close* from the form's System menu.

```
type TCloseQueryEvent = procedure(Sender: TObject; var CanClose: Boolean) of object;

property OnCloseQuery: TCloseQueryEvent;
```

Description Use *OnCloseQuery* to specify the conditions under which the form can close. An *OnCloseQuery* event handler contains a Boolean *CanClose* variable that determines whether a form is allowed to close. It's default value is *True*.

You can use an *OnCloseQuery* event handler to ask users if they are sure they really want the form closed immediately. For example, you can use the handler to display a message box that prompts the user to save a file before closing the form.

The *TCloseQueryEvent* type points to the method that determines whether a form can be closed. The value of the *CanClose* parameter determines if the form can close or not.

Example When the user attempts to close the form in this example, a message dialog appears that asks the user if it is OK to close the form. If the user chooses the OK button, the form closes. If the user chooses Cancel, the form doesn't close.

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if MessageDlg('Close the form?', mtConfirmation,
    [mbOk, mbCancel], 0) = mrCancel then
     CanClose := False;
end;
```

See also *OnClose* event

OnCreate

OnCreate occurs when the form is created.

type TNotifyEvent = **procedure**(Sender: TObject) **of object**;

property OnCreate: TNotifyEvent;

Description Use *OnCreate* to perform special processing when the form is created and is invoked by *TCustomForm*'s constructor. You should either implement this event or override the constructor of the *TCustomForm*; you should not do both. Any objects created in the *OnCreate* event should be freed by the *OnDestroy* event.

When a form is being created and its *Visible* property is True, the following events occur in the order listed:

- OnCreate
- OnShow
- OnActivate
- OnPaint

See also *OnDestroy* event, *OnActivate* event, *OnPaint* event, *OnShow* event

OnDeactivate

OnDeactivate occurs when the form loses focus.

type TNotifyEvent = **procedure**(Sender: TObject) **of object**;

```
property OnDeactivate: TNotifyEvent;
```

Description Use *OnDeactivate* to perform special processing when the form transitions from being the active form to another form in the same application becoming the active form. If activation goes to another application, this event is not triggered. To determine if another application has become active, Use the *TApplication* object's *OnDeactivate* event.

See also *OnActivate* event, *TApplication.OnDeactivate* event

OnDestroy

OnDestroy occurs when the form is destroyed.

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnDestroy: TNotifyEvent;
```

Description Use *OnDestroy* to perform special processing when the form is destroyed. You should either implement this event or override the destructor of the class, but not both. This event should destroy any objects created in the *OnCreate* event.

See also *OnCreate* event, *Free* method, *Destroy* method

OnHelp

OnHelp occurs when the form receives a request for help.

```
type THelpEvent = function (Command: Word; Data: Longint;  var CallHelp: Boolean): Boolean
  of object;

property OnHelp: THelpEvent;
```

Description Use *OnHelp* to write an event handler to perform special processing when help is requested.

See also *TApplication.HelpCommand* method, *TApplication.HelpContext* method, *HelpFile* property, *TApplication.HelpJump* method, *TApplication.HelpFile* property, *TApplication.OnHelp* event

OnHide

OnHide occurs when the form is hidden (for example, when the form's *Visible* property is set to *False*).

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnHide: TNotifyEvent;
```

Description Use *OnHide* to perform special processing when the form is hidden (for example, when the form's *Visible* property is set to *False*).

See also *OnShow* event

OnPaint

OnPaint occurs when the form is redrawn.

type TNotifyEvent = **procedure**(Sender: TObject) **of object**;

property OnPaint: TNotifyEvent;

Description Use *OnPaint* to perform special processing when the form is redrawn. Any special painting on the form should be done in this event. If you use the form's *Canvas* property outside the *OnPaint* event, it will be erased and drawn-over by the next *OnPaint* event.

Example The following code is an entire unit that loads a background bitmap onto the *Canvas* of the main form in the *OnPaint* event handler.

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
Dialogs;
type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    TheGraphic: TBitmap; { Add this declaration for the graphic}
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormPaint(Sender: TObject); { OnPaint event handler}
begin
  Form1.Canvas.Draw(0, 0, TheGraphic); { Draw the graphic on the Canvas }
end;
procedure TForm1.FormCreate(Sender: TObject); { OnCreate event handler }
begin
  TheGraphic := TBitmap.Create; { Create the bitmap object }
  TheGraphic.LoadFromFile('C:\APP\BKGRND.BMP'); { Load the bitmap from a file}
end;
end.
```

See also *Canvas* property

OnResize

OnResize occurs when the form is resized.

TNotifyEvent = **procedure**(Sender: TObject) **of object**;

property OnResize: TNotifyEvent;

Description Use *OnResize* to perform special processing when the form is resized.

See also *OnPaint* event

OnShow

OnShow occurs when the form is shown (for example, when the form's *Visible* property is set to *True*).

```
type TNotifyEvent = procedure(Sender: TObject) of object;

property OnShow: TNotifyEvent;
```

Description Use *OnShow* to perform special processing when the form is shown (for example, when the form's *Visible* property is set to *True*).

See also *OnHide* event

Methods

The following table lists the methods of *TCustomeForm:*

Method	Visibility	Documented in
Activate	protected	TCustomForm
ActiveChanged	protected	TCustomForm
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ChangeScale	protected	TCustomForm
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
Close	public	TCustomForm
CloseQuery	public	TCustomForm
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomForm
CreateNew	public	TCustomForm
CreateParams	protected	TCustomForm
CreateParented	public	TWinControl
CreateWindowHandle	protected	TCustomForm
CreateWnd	protected	TCustomForm

Method	Visibility	Documented in
Deactivate	protected	TCustomForm
DefaultHandler	protected	TCustomForm
DefineProperties	protected	TCustomForm
DefocusControl	public	TCustomForm
Destroy	public	TCustomForm
DestroyComponents	public	TComponent
Destroying	public	TComponent
DestroyWindowHandle	protected	TCustomForm
DisableAlign	public	TWinControl
DisableAutoRange	public	TScrollingWinControl
Dispatch	public	TObject
DoHide	protected	TCustomForm
DoShow	protected	TCustomForm
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EnableAutoRange	public	TScrollingWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TCustomForm
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetChildren	protected	TCustomForm
GetClientRect	protected	TCustomForm
GetFormImage	public	TCustomForm
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TCustomForm
InheritsFrom	public	TObject

Method	Visibility	Documented in
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
Loaded	protected	TCustomForm
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TCustomForm
Paint	protected	TCustomForm
PaintTo	public	TWinControl
PaintWindow	protected	TCustomForm
PaletteChanged	protected	TCustomForm
Perform	public	TControl
Print	public	TCustomForm
ReadState	protected	TCustomForm
Realign	public	TWinControl
Refresh	public	TControl
Release	public	TCustomForm
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
Resize	protected	TCustomForm
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
ScrollInView	public	TScrollingWinControl
SendCancelMode	public	TCustomForm
SendToBack	public	TControl
SetBounds	public	TWinControl
SetChildOrder	protected	TCustomForm
SetFocus	public	TCustomForm
SetFocusedControl	public	TCustomForm
SetParent	protected	TCustomForm
SetTextBuf	public	TControl
Show	public	TCustomForm
ShowModal	public	TCustomForm
Update	public	TWinControl
UpdateControlState	public	TWinControl
ValidateRename	protected	TCustomForm

Method	Visibility	Documented in
VisibleChanging	protected	TCustomForm
WndProc	protected	TCustomForm

Activate

The *Activate* method is the protected implementation method for the *OnActivate* event of form components.

```
procedure Activate; dynamic;
```

Description The form component calls *Activate* in response to the CM_ACTIVATE message, sent when a form gets the focus either because of a change in active windows within an application or because of the application becoming active.

The default *Activate* method calls any event handler attached by the user to the *OnActivate* event.

See also *Deactivate* method, *OnActivate* event

ActiveChanged

The ActiveChanged method is called when the active control on the form is changed.

```
procedure ActiveChanged; dynamic;
```

Description Implement this method in a *TCustomForm* descendant to execute some action when the active control on the form is changed.

See also *ActiveControl* property

C

ChangeScale

The *ChangeScale* method repositions and resizes the control by the ratio *M / D* and adjusts its client area and font size as needed.

```
procedure ChangeScale(M, D: Integer); override;
```

Description The *ScaleBy* method calls *ChangeScale* to perform its rescaling. *TWinControl* overrides *ChangeScale* to also rescale any controls for which the control is the parent before rescaling itself.

See also *ScaleBy* method

Close

Close closes a form.

```
procedure Close;
```

Description Use *Close* to close a form.

Close calls the *CloseQuery* method to determine if the form can close. If *CloseQuery* returns *False*, the close operation is aborted. Otherwise, the *OnClose* event occurs.

Note When the main form of the application is closed, the application terminates.

See also *CloseQuery* method

CloseQuery

CloseQuery returns a Boolean value specifying whether the form can close.

```
function CloseQuery: Boolean;
```

Description Use *CloseQuery* to determine whether the form can close. If the form can close, *CloseQuery* returns *True*; otherwise, *CloseQuery* returns *False*.

For an MDI parent form, *CloseQuery* calls the *CloseQuery* method on the form's MDI children to determine the return value.

See also *Close* method

Create

Create creates and initializes a new *TForm* object or an object derived from *TForm*.

```
constructor Create(AOwner: TComponent); override;
```

Description Use *Create* to create and initialize a new *TForm* object or an object derived from *TForm*. *AOwner* is the owner of the *TForm* object.

Create calls the *CreateNew* method, passing it *AOwner*, and then performs initialization tasks if the object is derived from *TForm*.

See also *CreateNew* method

CreateNew

CreateNew creates and initializes a new *TForm* object.

```
constructor CreateNew(AOwner: TComponent); override;
```

Description Use *CreateNew* to create and initialize a new *TForm* object.

CreateNew is called by the *Create* method. *AOwner* is the owner of the *TForm* object.

CreateNew performs the following tasks:

- Calls the *TScrollingWinControl.Create* method, passing it *AOwner*.
- Sets the *Left*, *Top*, *Width*, and *Height* properties to 0, 0, 320, and 240, respectively.

- Sets the *WindowState* property to wsNormal, the *Visible* property to *False*, and the *Ctrl3d* property to *True*, the *BorderIcons* property to [biSystemMenu, biMinimize, biMaximize], and the *BorderStyle* property to bsSizeable.
- Allocates memory for the *Icon* and *Canvas* properties.
- Sets the *Icon.OnChange* event to IconChanged.
- Sets the *TControl.ControlStyle* property to [csAcceptsControls, csCaptureMouse, csClickEvents, csSetCaption, csDoubleClicks].
- Sets the *Canvas.Control* property to Self.
- Sets the *PixelsPerInch* property to the value of the *Screen.PixelsPerInch* property and the *PrintScale* property to poProportional.
- Calls the *Screen.AddForm* method, passing it Self.

CreateNew should be used instead of *Create* when you want an object of the same class of the form but do not want the associated .DFM file to be used to initialize it. This might cause unpredictable results since most forms are written assuming they will have their controls created from the .DFM file.

See also *Create* method, *TScrollingWinControl.Create* method

CreateParams

The *CreateParams* method initializes a window-creation parameter record passed in the *Params* parameter.

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description The *CreateWnd* method calls *CreateParams* to initialize the parameters it passes to *CreateWindowHandle*. TCustomForm overrides CreateParams to customize the way the form creates its Windows representation. and calls the inherited method first. You should never need to override this method in the context of a form class.

The *TCreateParams* type is a data structure holding information needed when telling Windows to create a window handle. The fields of a *TCreateParams* record become the parameters to a call to the *CreateWindowEx* API function.

TWinControl implements *CreateParams* by setting all the fields of *Params* to generic base values.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
```

```
    WinClassName: array[0..63] of Char;
  end;
```

See also *CreateWindowHandle* method, *CreateWnd* method

CreateWindowHandle

The *CreateWindowHandle* method creates a window handle for the form.

```
procedure CreateWindowHandle(const Params: TCreateParams); override;
```

Description *CreateWindowHandle* is called to create a window handle for the form. *CreateWindowHandle* creates the window handle by calling the *CreateWindowEx* API function, passing parameters from the record passed in the *Params* parameter. *CreateWindowHandle* removes the CS_HREDRAW and CS_VREDRAW class styles from the window class.

The *TCreateParams* type is a data structure holding information needed when telling Windows to create a window handle.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;
```

See also *CreateParams* method, *CreateWnd* method

CreateWnd

The *CreateWnd* method creates a Windows control corresponding to the form.

```
procedure CreateWnd; override;
```

Description *CreateWnd* is called when the form is created, and whenever a change to the form requires the window to be recreated. *CreateWnd* first calls the inherited *CreateWnd*, and then sends the newly created window messages to request notification of Windows events, and, if the window is being recreated, reloads the form from a memory stream. *CreateWnd* is for internal use and you should never need to override it.

See also *CreateParams* method

Deactivate

The *Deactivate* method is the protected implementation method for the *OnDeactivate* event of form components.

```
procedure Deactivate; dynamic;
```

Description The form component calls *Deactivate* in response to the CM_DEACTIVATE message, sent when a form gets the focus either because of a change in active windows within an application or because of the application becoming active.

The default *Deactivate* method calls any event handler attached by the user to the *OnDeactivate* event.

See also *Activate* method, *OnActivate* event

DefaultHandler

The *DefaultHandler* method provides message handling for all messages that the form does not fully process itself.

```
procedure DefaultHandler(var Message); override;
```

Description Override *DefaultHandler* when you want to change the default message handling for the form. You should never need to override DefaultHandler since messages can be handled through creating a message method. See *ObjectPascal Language Guide* for more details.

Calling inherited in a message-handling method results in a call to the ancestor's *DefaultHandler* method if that ancestor does not specify a handler for the message being handled.

DefaultHandler passes any otherwise-unhandled messages to the form's window procedure by calling the *CallWindowProc* API function.

See also *Dispatch* method

DefineProperties

The *DefineProperties* method designates methods for storing a form's unpublished data on a stream such as a form file.

```
procedure DefineProperties(Filer: TFiler); override;
```

Description *DefineProperties* calls the inherited method and calls the filer object's *DefineProperty* method for the *PixelsPerInch* and *TextHeight* properties. *TForm* descendants can also specify methods that read and write data other than properties by overriding the *DefineProperties* method.

See also *TFiler.DefineProperty* method, *TFiler.DefineBinaryProperty* method

DefocusControl

DefocusControl removes focus from a control on the form.

```
procedure DefocusControl(Control: TWinControl; Removing: Boolean);
```

Description Use *DefocusControl* to remove focus from a control on the form. This method is used internally by the VCL.

Control is the control from which to remove focus. If *Control* is the form's active control, the form's *ActiveControl* property is set to **nil**.

Removing is a Boolean value specifying whether to set focus to the control's parent.

See also *SetFocusedControl* method, *ActiveControl* property, *SetFocus* method

Destroy

Destroy removes the form from memory.

```
destructor Destroy; override;
```

Description Do not call *Destroy*. Instead, call *Release*, which posts a CM_RELEASE message to itself. The handler for CM_RELEASE calls Free.

Destroy performs the following tasks:

- Frees the objects associated with the *Canvas*, *Icon*, and *Menu* properties.
- Calls the *OnDestroy* event handler for the object associated with the *OleFormObject* property (if the property is not **nil**).
- Frees memory associated with private objects.
- Calls the *TScrollingWinControl.Destroy* method.

See also *OnDestroy* event, *Release* method

DestroyWindowHandle

The *DestroyWindowHandle* method calls the Windows API function *DestroyWindow* to destroy the window handle created in the *CreateWindowHandle* method.

```
procedure DestroyWindowHandle; override;
```

Description The *Destroy* method calls *DestroyWindowHandle* to destroy any window handle associated with a form before destroying the form. The *DestroyWnd* method also calls *DestroyWindowHandle*.

See also *CreateWindowHandle* method, *DestroyWnd* method, *Destroy* method

DoHide

The *DoHide* method is the protected implementation method for the *OnHide* event of form components.

```
procedure DoHide; dynamic;
```

Description The form component calls *DoHide* in response to the CM_SHOWINGCHANGED message, sent when the application shows or hides the form.

The default *DoHide* method calls any event handler attached by the user to the *OnHide* event.

See also *OnHide* event, *DoShow* method, *Hide* method

DoShow

The *DoShow* method is the protected implementation method for the *OnShow* event of form components.

```
procedure DoShow; dynamic;
```

Description The form component calls *DoShow* in response to the CM_SHOWINGCHANGED message, sent when the application shows or hides the form.

The default *DoShow* method calls any event handler attached by the user to the *OnShow* event.

See also *DoHide* method, *OnShow* event

FocusControl

FocusControl sets focus to a control on the form.

```
procedure FocusControl(Control: TWinControl);
```

Description Use *FocusControl* to set focus to a control on the form.

Control is a *TWinControl* object encapsulating the control upon which to set focus. *FocusControl* sets the *ActiveControl* property.

See also *TWinControl* component, *ActiveControl* property

GetChildren

The *GetChildren* method returns the "child" components of the form; that is, those who return the form from their *GetParentComponent* method.

```
procedure GetChildren(Proc: TGetChildProc; Root: TComponent); override;
```

Description By default, *GetChildren* returns no children. For each child item, *GetChildren* should call the method passed in *Proc*. The children must be returned in creation order, the order they will appear in a form file.

Note *GetChildren* replaces the *WriteComponents* method used in Delphi 1.0. *GetChildren* supports form inheritance.

See also *GetParentComponent* method

GetClientRect

The *GetClientRect* method is a protected access method for the *ClientRect* property.

```
function GetClientRect: TRect; override;
```

Description *GetClientRect* returns a rectangle defining the rectangle inside the form's frame.

See also *GetClientOrigin* method

GetFormImage

GetFormImage is a bitmap of the form as it appears when printed.

```
function GetFormImage: TBitmap;
```

Description Use the *GetFormImage* method to obtain a bitmap of the form.

See also *Print* method

Loaded

Calls the inherited *Loaded* method and gives focus to the *ActiveControl* if assigned.

```
procedure Loaded; override;
```

Description First calls the inherited *TComponent.Loaded* method, and if the *ActiveControl* property of the form has been assigned, gives focus to the *ActiveControl*.

See also *ActiveControl* property, *TComponent.Loaded* method

Hide

Hide hides the form.

```
procedure Hide;
```

Description *Hide* sets the *Visible* property to *False*.

See also *Visible* property

Notification

The *Notification* method notifies the form that the component specified by *AComponent* is about to be inserted or removed, as specified by *Operation*.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description By default, forms pass along the notification to their owned components, if any.

A form can, if needed, act on the notification that a component is being inserted or removed. In particular, if a form has object fields or properties that contains references to other components, it might check the notifications of component removals and invalidate those references as needed.

See also *InsertComponent* method

Paint

The *Paint* method calls the method associated with the *OnPaint* event of the form.

```
procedure Paint; dynamic;
```

Description Override *Paint* to change the way the form is drawn.

See also *OnPaint* event

PaintWindow

The *PaintWindow* method repaints the contents of the form.

```
procedure PaintWindow(DC: HDC); override;
```

Description *PaintWindow* sends a WM_PAINT message to the form's *DefaultHandler* message, setting the message record's *WParam* field to the value passed in *DC* and the other parameter and result fields to zeros.

See also *DefaultHandler* method

PaletteChanged

The *PaletteChanged* method responds to changes in the system's palette by realizing the control's palette as returned from *GetPalette*.

```
function PaletteChanged(Foreground: Boolean): Boolean; override;
```

Description You should never need to alter the default behavior of *PaletteChanged*. The *Foreground* parameter indicates whether the palette realization should treat this control's palette as being in the foreground or the background.

The return value from *PaletteChanged* indicates whether the form actually realized its palette, returning *True* only if *GetPalette* returned a valid palette.

See also *GetPalette* method

Print

Print prints the form.

```
procedure Print;
```

Description Use the *Print* method to print the form

See also *GetFormImage* method

ReadState

The *ReadState* method reads the value of the form's published properties, stored data, and owned components. from the reader object passed in *Reader*.

```
procedure ReadState(Reader: TReader); override;
```

Description The *ReadState* method reads the value of the form's published properties, stored data, and owned components from the reader object passed in *Reader*.

See also *TReader* object

Release

Release destroys the form and frees its associated memory.

```
procedure Release;
```

Description Use *Release* to destroy the form and free its associated memory.

Release is much like the *Free* method except that *Release* does not destroy the form until all event handlers of the form and event handlers of components on the form have finished executing. Any event handlers of the form should use *Release* instead of *Free*. Failing to do so could lead to an access violation exception being generated.

See also *Free* method, *Destroy* method

Resize

The *Resize* method is the protected implementation method for the *OnResize* event of scroll boxes, forms, and panels.

```
procedure Resize; dynamic;
```

Description The form calls *Resize* in response to the WM_SIZE message, indicating a change in the size of the form.

By default, *Resize* does nothing except call any event handler attached to the *OnResize* event. You can override *Resize* to provide other responses in addition to the inherited event-handler call.

See also *OnResize* event

SendCancelMode

SendCancelMode cancels modes on the form.

```
procedure SendCancelMode(Sender: TControl);
```

Description Use *SendCancelMode* to release the mouse capture and cancel scroll bar input and menu processing on the form.

See also *ActiveControl* property

SetChildOrder

The *SetChildOrder* method changes the order in which *Child* appears when *GetChildren* is called.

```
procedure SetChildOrder(Child: TComponent; Order: Integer); override;
```

Description The *Child* moves as if it were an item in a list object: items previously below the child's old position move up, and those below the new position move down.

See also *GetChildren* method

SetFocus

SetFocus sets focus to the form.

```
procedure SetFocus;
```

Description Use *SetFocus* to set focus to the form. *SetFocus* calls the *SetFocus* method of the active control, if one exists.To determine whether the form has focus, use the *Active* property.

See also *Active* property, *SetFocusedControl* method

SetFocusedControl

SetFocusedControl sets focus to a control on the form.

```
function SetFocusedControl(Control: TWinControl): Boolean;
```

Description Use *SetFocusedControl* to set focus to a control on the form.

See also *Active* property, *SetFocus* method, *DefocusControl* method

SetParent

SetParent calls the control's *InsertControl* method to add the control to the form's list of controls.

```
procedure SetParent(AParent: TWinControl); override;
```

Description *SetParent* calls the inherited *SetParent* method if it is valid to add *AParent* to the form's list of controls.

See also *Parent* property

Show

Show shows the form.

```
procedure Show;
```

Description Use *Show* to set the form's *Visible* property to *True* and to bring the form to the front of other forms on the screen.

See also *BringToFront* method, *Visible* property

ShowModal

ShowModal shows a form as a modal form.

```
function ShowModal: Integer;
```

Description Use *ShowModal* to show a form as a modal form, which means the user must put the form away before the application can continue to run.

To close a modal form, set its *ModalResult* property to a nonzero value.

ShowModal returns the *ModalResult* property of the form.

See also *ModalResult* property

ValidateRename

The *ValidateRename* method determines if a form can rename one of its owned components.

```
procedure ValidateRename(AComponent: TComponent; const CurName, NewName: string);
override;
```

Description The *ValidateRename* method checks to see if a form can rename one of its owned components, passed in *AComponent*, from its current name (*CurName*) to the string passed in *NewName*. If *AComponent* is **nil** or *NewName* is already the name of a component in the form's list, *ValidateRename* raises an *EComponentError* exception.

See also *FindComponent* method

VisibleChanging

The *VisibleChanging* method is a protected method called just before the form changes the value of its *Visible* property.

```
procedure VisibleChanging; override;
```

Description If, for some reason, the form should not change its visible state, it should raise an exception.

See also *Visible* property

WndProc

The *WndProc* method provides specific message responses for the form. *WndProc* is the first method that receives messages for a Delphi form.

```
procedure WndProc(var Message: TMessage); override;
```

Description *WndProc* calls the inherited *WndProc* method. WndProc performs some window activation, focus, and position message handling necessary to keep VCL in sync with Windows. Override *WndProc* to change how the control responds to Windows messages.

See also *TWinControl.WndProc* method

TCustomGrid

Unit: Grids

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomGrid

TCustomGrid is the abstract base type for all components that present information in a two-dimensional grid.

Description Use *TCustomGrid* as a base class when defining components that represent information in a tabular format. *TCustomGrid* introduces properties and methods to control the appearance and behavior of the grid, including responses to mouse actions and user editing.

Do not create instances of *TCustomGrid*. Instead, put a grid control in a form by instantiating a descendant of *TCustomGrid*. Use *TDrawGrid* to represent non-textual material in a tabular grid. For text, use a *TStringGrid* object. If the text represents field values from a dataset, use *TDBGrid*.

See also *TDBGrid* object, *TDrawGrid* object, *TStringGrid* object

Properties

The following table lists the properties of *TCustomGrid*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BorderStyle	protected	bsSingle	TCustomGrid
BoundsRect	public		TControl

Property	Visibility	Default	Documented in
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Col	protected		TCustomGrid
ColCount	protected	5	TCustomGrid
Color	protected	clWindow	TControl
ColWidths	protected		TCustomGrid
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DefaultColWidth	protected	64	TCustomGrid
DefaultDrawing	protected	True	TCustomGrid
DefaultRowHeight	protected	24	TCustomGrid
DesignInfo	public		TComponent
EditorMode	protected		TCustomGrid
Enabled	public	True	TControl
FixedColor	protected	clBtnFace	TCustomGrid
FixedCols	protected	1	TCustomGrid
FixedRows	protected	1	TCustomGrid
GridHeight	protected, read-only		TCustomGrid
GridLineWidth	protected	1	TCustomGrid
GridWidth	protected, read-only		TCustomGrid
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
HitTest	protected, read-only		TCustomGrid
InplaceEditor	protected, read-only		TCustomGrid
Left	published		TControl
LeftCol	protected		TCustomGrid
Name	published		TControl

Property	Visibility	Default	Documented in
Options	protected	[goFixedVertLine, goFixedHorzLine, goVertLine, goHorzLine, goRangeSelect]	TCustomGrid
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	False	TControl
ParentWindow	public		TWinControl
Row	protected		TCustomGrid
RowCount	protected	5	TCustomGrid
RowHeights	protected		TCustomGrid
ScrollBars	protected	ssBoth	TCustomGrid
Selection	protected		TCustomGrid
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
TabStops	protected		TCustomGrid
Tag	published	0	TComponent
Top	published		TControl
TopRow	protected		TCustomGrid
VCLComObject	public		TComponent
Visible	public	True	TControl
VisibleColCount	protected, read-only		TCustomGrid
VisibleRowCount	protected, read-only		TCustomGrid
Width	published		TControl
WindowProc	public		TControl

C

BorderStyle

BorderStyle determines whether a single line border is drawn around the grid.

property BorderStyle: TBorderStyle;

Description Set *BorderStyle* to *bsSingle* to add a single line border around the grid's image. Set *BorderStyle* to *bsNone* to omit the border.

The visual effect of *BorderStyle* is influenced by the value of the *Ctrl3D* property. Experiment with different combinations of *BorderStyle* and *Ctrl3D* to achieve the desired look.

See also *TWinControl.Ctrl3D* property

Col

Col is the index of the column that contains the selected cell.

property Col: Longint;

Description Use *Col* at runtime to determine the current column in the grid. Setting *Col* moves focus to the cell in the current row that is in the new column. The first column has an index of 0, the second column an index of 1, and so on.

The selected cell in the grid can be located by reading the *Col* property and the *Row* property to obtain the indexes of its column and row. When focus moves to the grid, the selected cell gets input focus.

See also *Row* property

ColCount

ColCount is the number of columns in the grid.

property ColCount: Longint;

Description Read *ColCount* to determine the number entries in the *ColWidths* array. Set *ColCount* to add or delete columns at the right side of the grid. The value of *ColCount* includes any fixed columns at the left of the grid as well as the scrollable columns in the body of the grid.

See also *ColWidths* property, *RowCount* property, *VisibleColCount* property

ColWidths

ColWidths is an indexed array giving the width in pixels of all the columns in the grid.

property ColWidths[Index: Longint]: Integer;

Description Read *ColWidths* to determine the width of the cells in the column specified by the *Index* parameter. An index of 0 refers to the first column. Set *ColWidths* at runtime to change the width of an individual column.

If the width of a column has not been set explicitly by resizing with the mouse, or by using the *ColWidths* property, it's width is *DefaultColWidth*.

Use the *ColCount* property to determine the number of entries in the *ColWidths* array.

See also *ColCount* property, *DefaultColWidth* property, *RowHeights* property

DefaultColWidth

DefaultColWidth is the width in pixels of all columns in the grid that have not been explicitly resized.

property DefaultColWidth: Integer;

Description Set *DefaultColWidth* to change the size of all columns in the grid. When *DefaultColWidth* is set, columns that have been resized using the mouse or by setting the *ColWidths* property are given the *DefaultColWidth* as well.

When new columns are added to the grid using the *ColCount* property, they are created with a width of *DefaultColWidth*.

See also *ColCount* property, *ColWidths* property, *DefaultRowHeight* property

DefaultDrawing

DefaultDrawing determines whether the *Cells* are painted when the grid is drawn.

```
property DefaultDrawing: Boolean;
```

Description Set *DefaultDrawing* to *True* to have the grid automatically paint the cells in the grid, including adding the background color, the 3-D effects on the fixed cells, and the focus rectangle around the selected cell. Set *DefaultDrawing* to *False* to skip the painting of the cells when the grid is drawn.

When *DefaultDrawing* is *False*, the cells should be painted using the *DrawCell* method. For *TCustomGrid*, *DrawCell* is an abstract method. Descendants of *TCustomGrid* use *DrawCell* to provide their own customized drawing code, or to generate an *OnDrawCell* event.

See also *DrawCell* method, *TDrawGrid.OnDrawCell* event

DefaultRowHeight

DefaultRowHeight is the height in pixels of all rows in the grid that have not been explicitly resized.

```
property DefaultColWidth: Integer;
```

Description Set *DefaultRowHeight* to change the size of all rows in the grid. When *DefaultRowHeight* is set, rows that have been resized using the mouse or by setting the *RowHeights* property are given the *DefaultRowHeight* as well.

When new rows are added to the grid using the *RowCount* property, they are created with a height of *DefaultRowHeight*.

See also *DefaultColWidth* property, *RowCount* property, *RowHeights* property

EditorMode

EditorMode determines whether the current cell can be edited.

```
property EditorMode: Boolean;
```

Description Set *EditorMode* to *True*, at runtime, to put the grid in edit mode. When *EditorMode* is *True*, the user can edit cells in the grid. When the user presses *F2*, *EditorMode* is set to *True*. When the user presses *Enter*, the value of *EditorMode* is toggled.

Note *EditorMode* has no effect if the *Options* property does not include *goEditing* or if the *Options* property includes *goAlwaysShowEditor*.

See also *HideEditor* method, *Options* property, *ShowEditor* method

FixedColor

FixedColor is the background color of the fixed rows and columns in the grid.

```
property FixedColor: TColor;
```

Description Set *FixedColor* to specify the color for the nonscrolling cells in the grid. The scrolling region of the grid is drawn using the *Color* property.

The possible values for *FixedColor* are listed under the entry for the *Color* property.

See also *Color* property, *FixedCols* property, *FixedRows* property

FixedCols

FixedCols is the number of columns on the left of the grid that cannot be scrolled.

```
property FixedCols: Integer;
```

Description Set *FixedCols* to create or get rid of nonscrolling columns. Nonscrolling columns appear at the left of the grid, and are always visible, even when the user scrolls the other columns in the grid. Use nonscrolling columns for displaying row titles or row numbers, or to implement a scroll lock that the user can set.

Nonscrolling columns are visually distinct from the scrollable columns in the grid. They are painted with a separate background color, and their cells have a raised, 3-D look when the *Ctrl3D* property is *True*.

Note Grids must include at least one scrolling column. Do not set *FixedCols* to a value greater than *ColCount* – 1.

See also *ColCount* property, *DefaultDrawing* property, *FixedColor* property, *FixedRows* property, *TWinControl.Ctrl3D* property

FixedRows

FixedRows is the number of rows on the left of the grid that cannot be scrolled.

```
property FixedRows: Integer;
```

Description Set *FixedRows* to create or get rid of nonscrolling rows. Nonscrolling rows appear at the top of the grid, and are always visible, even when the user scrolls the other rows in the grid. Use nonscrolling rows for displaying column titles or column numbers.

Nonscrolling rows are visually distinct from the scrollable rows in the grid. They are painted with a separate background color, and their cells have a raised, 3-D look when the *Ctrl3D* property is *True*.

Note Grids must include at least one scrolling row. Do not set *FixedRows* to a value greater than *RowCount* – 1.

See also *DefaultDrawing* property, *FixedColor* property, *FixedCols* property, *RowCount* property, *TWinControl.Ctrl3D* property

GridHeight

GridHeight is the height of the grid in pixels.

property GridHeight: Integer;

Description Read *GridHeight* to determine the size of the grid. If *GridHeight* is less than the value of *ClientHeight*, all of the rows of the grid appear in the control with an empty region below the grid. If the underlying grid is too tall to appear in the control, *GridHeight* is the same as *ClientHeight*, and the user must scroll to see the entire contents of the grid.

See also *ClientHeight* property, *GridWidth* property

GridLineWidth

GridLineWidth is the width, in pixels, of the lines that separate the cells of the grid.

property GridLineWidth: Integer;

Description Set *GridLineWidth* to make the lines that separate the cells in the grid heavier or lighter. When *GridLineWidth* is zero, no separators are drawn between the cells of the grid.

GridLineWidth will have no effect if the *Options* property does not include *goFixedHorzLine*, *goFixedVertLine*, *goHorzLine*, or *goVertLine*.

Note Even when *GridLineWidth* is zero, the cells in the fixed rows and columns of the grid will appear distinct if the *Ctrl3D* property is *True*. Set *Ctrl3D* to *False* to remove the raised effect that separates the cells in the fixed region of the grid.

Note *Values* greater than 3 pixels are not recommended for applications that will run on Windows 95 because of distortions that can appear.

See also *FixedCols* property, *FixedRows* property, *Options* property, *TWinControl.Ctrl3D* property

GridWidth

GridWidth is the width of the grid in pixels.

property GridWidth: Integer;

Description Read *GridWidth* to determine the size of the grid. If *GridWidth* is less than the value of *ClientWidth*, all of the columns of the grid appear in the control with an empty region to the right of the grid. If the underlying grid is too wide to appear in the control, *GridWidth* is the same as *ClientWidth*, and the user must scroll to see the entire contents of the grid.

See also *ClientWidth* property, *GridHeight* property

HitTest

HitTest indicates the current position of the mouse cursor.

```
property HitTest: TPoint;
```

Description Read *HitTest* to determine the position of the mouse when it is over the grid or when the grid has captured the mouse. *HitTest* may represent a position that is not within the client area of the grid control if the user is moving or resizing the rows or columns, or if the user is selecting a range of cells using the mouse. *HitTest* is the position of the mouse cursor in the coordinate system where (0,0) is the upper left corner of the grid control's client area.

See also *ClientOrigin* property, *MouseCoord* method

InplaceEditor

InplaceEditor is the edit control the grid uses to allow users to edit the contents of the selected cell.

```
property InplaceEditor: TInplaceEdit;
```

Description Read *InplaceEditor* to gain direct access to the edit control that sits on top of the selected cell when the grid is in edit mode. The in-place editor is created the first time the grid is put in edit mode. It is not destroyed until the grid is destroyed. The grid moves the editor when the selected cell changes. The grid hides the editor's window when the grid goes out of edit mode.

See also *CreateEditor* method, *EditorMode* property, *HideEditor* method, *ShowEditor* method

LeftCol

LeftCol is the number of the first visible scrollable column in the grid.

```
property LeftCol: Longint;
```

Description Read *LeftCol* to determine the number of the first column in the scrollable region that is visible. Set *LeftCol* to scroll the columns in the grid so that the column with index *LeftCol* is the first column after the fixed columns.

See also *FixedCols* property, *ScrollData* method, *VisibleColCount* property

Options

Options specifies various display and behavioral properties of the grid.

```
type
  TGridOption = (goFixedVertLine, goFixedHorzLine, goVertLine, goHorzLine,
    goRangeSelect, goDrawFocusSelected, goRowSizing, goColSizing, goRowMoving,
    goColMoving, goEditing, goTabs, goRowSelect, goAlwaysShowEditor, goThumbTracking);
  TGridOptions = set of TGridOption;

property Options: TGridOptions;
```

Description Set *Options* to include the desired properties for the grid. *Options* is a set drawn from the following values:

Value	Meaning
goFixedVertLine	Vertical lines are drawn to separate the fixed (nonscrolling) columns in the grid.
goFixedHorzLine	Horizontal lines are drawn to separate the fixed (nonscrolling) rows in the grid.
goVertLine	Vertical lines are drawn to separate the scrollable columns in the grid.
goHorzLine	Horizontal lines are drawn to separate the scrollable rows in the grid.
goRangeSelect	Users can select ranges of cells at one time. *goRangeSelect* is ignored if *Options* includes *goEditing*.
goDrawFocusSelected	Cells with input focus are drawn the with a special highlight color, just like selected cells without input focus. If *goDrawFocusSelected* is not included, the cell with input focus is distinguished by a focus rectangle, not by a special background color.
goRowSizing	Scrollable rows can be individually resized.
goColSizing	Scrollable columns can be individually resized.
goRowMoving	Scrollable rows can be moved using the mouse.
goColMoving	Scrollable columns can be moved using the mouse.
goEditing	Users can edit the contents of cells. When *goEditing* is included in *Options*, *goRangeSelect* has no effect.
goTabs	Users can navigate through the cells in the grid using *Tab* and *Shift+Tab*.
goRowSelect	Entire rows are selected rather than individual cells. If *goRowSelect* is included in *Options*, *goAlwaysShowEditor* has no effect.
goAlwaysShowEditor	The grid is locked into edit mode. The user does not need to use *Enter* or *F2* to turn on *EditorMode*. If *Options* does not include *goEditing*, *goAlwaysShowEditor* has no effect. If *Options* includes *goRowSelect*, *goAlwaysShowEditor* has no effect.
goThumbTracking	The grid image updates while the user is dragging the thumb of the scrollbar. If *goThumbTracking* is not included, the image does not update until the user releases the thumb in a new position.

C

See also *ColWidths* property, *EditorMode* property, *GridLineWidth* property, *RowHeights* property, *Selection* property, *TabStops* property

Row

Row is the index of the row that contains the selected cell.

property Row: Longint;

Description Use *Row* at runtime to determine the current row in the grid. Setting *Row* moves focus to the cell in the current column that is in the new row. The first row has an index of 0, the second row an index of 1, and so on.

The selected cell in the grid can be located by reading the *Row* property and the *Col* property to obtain the indexes of its row and column. When focus moves to the grid, the selected cell gets input focus.

See also *Column* property

RowCount

RowCount is the number of rows in the grid.

property RowCount: Longint;

Description Read *RowCount* to determine the number entries in the *RowHeights* array. Set *RowCount* to add or delete rows at the bottom of the grid. The value of *RowCount* includes any fixed rows at the top of the grid as well as the scrollable rows in the body of the grid.

See also *ColCount* property, *RowHeights* property, *VisibleRowCount* property

RowHeights

RowHeights is an indexed array giving the height in pixels of all the rows in the grid.

property RowHeights[Index: Longint]: Integer;

Description Read *RowHeights* to determine the height of the cells in the row specified by the *Index* parameter. An index of 0 refers to the first row. Set *RowWidths* at runtime to change the height of an individual row.

If the height of a row has not been set explicitly by resizing with the mouse, or by using the *RowHeights* property, it's height is *DefaultRowHeight*.

Use the *RowCount* property to determine the number of entries in the *RowHeights* array.

See also *ColWidths* property, *DefaultRowHeight* property, *RowCount* property

ScrollBars

ScrollBars specifies whether the grid includes horizontal or vertical scroll bars.

```
type TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth);

property ScrollBars: TScrollStyle;
```

Description Use *ScrollBars* to give the grid horizontal or vertical scroll bars. *ScrollBars* can take one of the following values:

Value	Meaning
ssNone	The grid has no scroll bars.
ssHorizontal	The grid has a single scroll bar on the bottom edge.
ssVertical	The grid has a single scroll bar on the right edge.
ssBoth	The grid has a scroll bar on both the bottom and right edges.

Note If all the cells in the grid fit in the *ClientWidth*, no horizontal scrollbar appears, even if *ScrollBars* is *ssHorizontal* or *ssBoth*. If all the cells fit in the *ClientHeight*, no vertical scrollbar appears, even if *ScrollBars* is *ssVertical* or *ssBoth*.

See also *ClientHeight* property, *ClientWidth* property

Selection

Selection indicates the boundaries of the current selection.

```
TGridRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Longint);
    1: (TopLeft, BottomRight: TGridCoord);
  end;

property Selection: TGridRect;
```

Description Set *Selection* to select a range of cells in the grid. Selection can either represent the first column, first row, last column and last row in the grid, or the row and column coordinates of the upper left and bottom right cells in the selected region.

Selection can only indicate more than one cell when *Options* includes *goRangeSelect*.

See also *Col* property, *Options* property, *Row* property

TabStops

TabStops is an indexed array of Boolean values that indicate whether the user can tab to specified columns in the grid.

property TabStops[Index: Longint]: Boolean;

Description Set *TabStops* to *False* to remove the column identified by *Index* from the tab order. The first column in the grid is identified by an *Index* of 0.

Note When the *Options* property does not include *goTabs*, the user can't tab to any column, even if *TabStops* is *True*.

See also *Options* property

TopRow

TopRow is the index of the first visible scrollable row in the grid.

property TopRow: Longint;

Description Read *TopRow* to determine the index of the first row in the scrollable region that is visible. Set *TopRow* to scroll the rows in the grid so that the row with index *TopRow* is the first row after the fixed rows.

See also *FixedRows* property, *ScrollData* method, *VisibleRowCount* property

VisibleColCount

VisibleColCount is the number of columns visible in the grid.

property VisibleColCount: Integer;

Description Use *VisibleColCount* to determine the number of scrollable columns fully visible in the grid. *VisibleColCount* does not include the fixed columns counted by the *FixedCols* property. It does not include any partially visible columns on the right edge of the grid.

See also *ColCount* property, *FixedCols* property, *VisibleRowCount* property

VisibleRowCount

VisibleRowCount is the number of rows visible in the grid.

property VisibleColCount: Integer;

Description Use *VisibleRowCount* to determine the number of scrollable rows fully visible in the grid. *VisibleRowCount* does not include the fixed rows counted by the *FixedRows* property. It does not include any partially visible rows on the bottom of the grid.

See also *FixedRows* property, *RowCount* property, *VisibleColCount* property

Methods

The following table lists the methods of *TCustomGrid*:

Method	Visibility	Documented in
AdjustSize	protected	TCustomGrid
Assign	public	TPersistent
BeginDrag	public	TControl
BoxRect	protected	TCustomGrid
BringToFront	public	TControl
Broadcast	public	TWinControl
CanEditAcceptKey	protected	TCustomGrid
CanEditModify	protected	TCustomGrid
CanEditShow	protected	TCustomGrid
CanFocus	public	TWinControl
CanGridAcceptKey	protected	TCustomGrid
CellRect	protected	TCustomGrid
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ColumnMoved	protected	TCustomGrid
ColWidthsChanged	protected	TCustomGrid
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomGrid
CreateEditor	protected	TCustomGrid
CreateParams	protected	TCustomGrid
CreateParented	public	TWinControl
DefaultHandler	public	TObject
DefineProperties	protected	TCustomGrid
DeleteColumn	protected	TCustomGrid
DeleteRow	protected	TCustomGrid
Destroy	public	TCustomGrid
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DoExit	protected	TCustomGrid
DragDrop	public	TControl

Method	Visibility	Documented in
Dragging	public	TControl
DrawCell	protected	TCustomGrid
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetEditLimit	protected	TCustomGrid
GetEditMask	protected	TCustomGrid
GetEditText	protected	TCustomGrid
GetGridHeight	protected	TCustomGrid
GetGridWidth	protected	TCustomGrid
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
HideEditor	protected	TCustomGrid
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
InvalidateCell	protected	TCustomGrid
InvalidateCol	protected	TCustomGrid
InvalidateEditor	protected	TCustomGrid
InvalidateRow	protected	TCustomGrid
KeyDown	protected	TCustomGrid
KeyPress	protected	TCustomGrid
MethodAddress	public	TObject

Method	Visibility	Documented in
MethodName	public	TObject
MouseCoord	public	TCustomGrid
MouseDown	protected	TCustomGrid
MouseMove	protected	TCustomGrid
MouseUp	protected	TCustomGrid
MoveColRow	protected	TCustomGrid
MoveColumn	protected	TCustomGrid
MoveRow	protected	TCustomGrid
NewInstance	public	TObject
Paint	protected	TCustomGrid
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
RowHeightsChanged	protected	TCustomGrid
RowMoved	protected	TCustomGrid
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
ScrollData	protected	TCustomGrid
SelectCell	protected	TCustomGrid
SendToBack	public	TControl
SetBounds	public	TWinControl
SetEditText	protected	TCustomGrid
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
ShowEditor	protected	TCustomGrid
ShowEditorChar	protected	TCustomGrid
SizeChanged	protected	TCustomGrid
Sizing	protected	TCustomGrid
TimedScroll	protected	TCustomGrid
TopLeftChanged	protected	TCustomGrid
Update	public	TWinControl
UpdateControlState	public	TWinControl
UpdateDesigner	protected	TCustomGrid

AdjustSize

AdjustSize adds or deletes rows or columns from the grid.

```
procedure AdjustSize(Index, Amount: Longint; Rows: Boolean); dynamic;
```

Description Grid components call *AdjustSize* internally to fix up the number of rows or columns when the user resizes them using the mouse. *Index* is the index of the row or column where the insertion or deletion should begin. *Amount* is the number of rows or columns to insert (*Amount* > 0) or delete (*Amount* < 0). *Rows* is a boolean that indicates whether the rows or the columns are changing.

See also *DefaultColWidth* property, *DefaultRowHeight* property

BoxRect

BoxRect returns the screen coordinates of a specified set of cells.

```
function BoxRect(ALeft, ATop, ARight, ABottom: Longint): TRect;
```

Description Call *BoxRect* to obtain the screen coordinates, in pixels, of the cells that run from the column with index *ALeft* to the column with index *ARight* and the row with index *ATop* to the row with index *ABottom*. The coordinates of cells that are not visible onscreen are not included in the final rectangle.

See also *BoundsRect* property, *CellRect* method

CanEditAcceptKey

CanEditAcceptKey indicates whether the *Key* parameter can be entered into the current cell.

```
function CanEditAcceptKey(Key: Char): Boolean; dynamic;
```

Description *CanEditAcceptKey* provides a simple screening of keyboard input while the user is editing the contents of a cell. The *CanEditAcceptKey* method for *TCustomGrid* always returns *True*. Descendants of *TCustomGrid* override this method to screen out illegal characters.

More refined validation can be obtained by using an edit mask.

See also *CanGridAcceptKey* method, *GetEditMask* method, *KeyPress* method

CanEditModify

CanEditModify indicates whether the current cell in the grid can be edited.

```
function CanEditModify: Boolean; dynamic;
```

Description Grid components use *CanEditModify* internally to determine whether to allow the user to edit the current cell. *CanEditModify* returns *True* unless the inplace editor has disallowed editing for the cell.

See also *CanEditShow* method, *InplaceEditor* property

CanEditShow

CanEditShow indicates whether the inplace edit control can be created to allow editing.

```
function CanEditShow: Boolean; dynamic;
```

Description Grid controls call *CanEditShow* internally before creating the editor for a cell. *CanEditShow* is called after the grid has determined that the field can be edited by calling *CanEditModify*.

CanEditShow returns *False* if the *Options* property does not indicate that the grid allows editing, or if it indicates that the grid permits row selection. Otherwise, if the *Options* property indicates that the grid is locked in edit mode, or if the grid has input focus, *CanEditShow* returns *True*. If the grid is not locked in edit mode and the grid does not have input focus, *CanEditShow* returns *False*.

See also *CanEditModify* method, *InplaceEditor* property, *Options* property

CanGridAcceptKey

CanGridAcceptKey indicates whether the grid should respond to a given key combination.

```
TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

function CanGridAcceptKey(Key: Word; Shift: TShiftState): Boolean; dynamic;
```

Description Grid controls call *CanGridAcceptKey* internally to screen key combinations when the user first presses a key. The *CanGridAcceptKey* method for *TCustomGrid* always returns *True*. Descendants of *TCustomGrid* override this method to screen out irrelevant key combinations, or key combinations that indicate actions which would violate the *Options* or mode of the grid.

See also *CanEditAcceptKey* method, *KeyDown* method, *Options* property

CellRect

CellRect returns the screen coordinates of a cell in the grid.

```
function CellRect(ACol, ARow: Longint): TRect;
```

Description Call *CellRect* to obtain the boundaries, in pixels, of the cell in the column and row indicated by the *ACol* and *ARow* parameters. The top left cell is specified when *ACol* and *ARow* are set to zero.

If the indicated cell is not visible, *CellRect* returns an empty rectangle.

See also *BoxRect* method

ColumnMoved

ColumnMoved provides the interface for a method that responds when the position of a column changes.

procedure ColumnMoved(FromIndex, ToIndex: Longint); **dynamic**;

Description *ColumnMoved* is called immediately after a column in the grid has moved. The *FromIndex* parameter is the old index of the column, and the *ToIndex* parameter is the new index. The *ColumnMoved* method of *TCustomGrid* does nothing. Descendants of *TCustomGrid* override this method to make internal adjustments or to generate an *OnColumnMoved* event.

See also *MoveColumn* method

ColWidthsChanged

ColWidthsChanged responds when the column widths change.

procedure ColWidthsChanged; **dynamic**;

Description *ColWidthsChanged* is called immediately after the column widths change. The change can result from setting the *ColWidths* property, from setting the *DefaultColWidth* property, from moving one of the columns, or from resizing the columns with the mouse.

ColWidthsChanged adjusts the scroll bar and the position of the inplace editor to take into account the new column size and positions of the columns.

See also *ColWidths* property, *DefaultColWidth* property, *InplaceEditor* property, *ScrollBars* property

Create

Create creates an instance of *TCustomGrid*.

constructor Create(AOwner: TComponent); **override**;

Description Do not directly create instances of *TCustomGrid*. *TCustomGrid* is an abstract base class for grid controls. Instead, call the *Create* method of a descendant of *TCustomGrid* to create a grid control at runtime. Grid controls placed on a form at design time are created automatically.

After calling the inherited constructor, *Create*

- Initializes the *BorderStyle* property to *bsSingle*.
- Initializes the *ColCount* and *RowCount* properties to 5.
- Initializes the *Color* property to *clWindow* and the *FixedColor* property to *clBtnFace*.
- Initializes the *ControlStyle* property to [*csCaptureMouse, csOpaque, csDoubleClicks, csFramed*]

- Initializes the *DefaultColWidth* property to 64 and the *DefaultRowHeight* property to 24.
- Initializes the *DefaultDrawing* property to *True*.
- Initializes the *EditorMode* property to *False*.
- Initializes the *FixedCols* and *FixedRows* properties to 1.
- Initializes the *GridLineWidth* property to 1.
- Initializes the *Options* property to [*goFixedVertLine, goFixedHorzLine, goVertLine, goHorzLine, goRangeSelect*]
- Initializes the *ParentColor* property to *False*.
- Initializes the *ScrollBars* property to *ssBoth*.
- Initializes the *TabStop* property to *True*.

See also *BorderStyle* property, *ColCount* property, *Color* property, *ControlStyle* property, *DefaultColWidth* property, *DefaultRowHeight* property, *DefaultDrawing* property, *EditorMode* property, *FixedColor* property, *FixedCols* property, *FixedRows* property, *GridLineWidth* property, *Options* property, *ParentColor* property, *RowCount* property, *ScrollBars* property, *TabStop* property, *TWinControl.Create* method

CreateEditor

CreateEditor creates the inplace editor control.

```
function CreateEditor: TInplaceEdit; virtual;
```

Description Grid controls call *CreateEditor* internally the first time the grid goes into edit mode. *CreateEditor* is exposed as a protected method so that descendants of *TCustomGrid* can substitute a specialized descendant of *TInplaceEdit* to handle cell editing.

See also *EditorMode* property, *InplaceEditor* property, *ShowEditor* method

CreateParams

CreateParams initializes a window-creation parameter record passed in the *Params* parameter.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
```

```
    WinClassName: array[0..63] of Char;
  end;

procedure CreateParams(var Params: TCreateParams); override;
```

Description The *CreateWnd* method is called internally to specify the properties of the window that implements this control. The *TCreateParams* type is a data structure holding information needed when telling Windows to create a window handle. The fields of a *TCreateParams* record become the parameters to a call to the *CreateWindowEx* API function.

After calling the inherited method, *CreateParams* adjusts the value of *Params* to request that the grid receive double click messages, and to implement the values of the *Ctrl3D*, *ScrollBars*, and *BorderStyle* properties.

See also *BorderStyle* property, *ScrollBars* property, *TWinControl.CreateWnd* method, *TWinControl.Ctrl3D* property

DefineProperties

DefineProperties adds the *ColWidths* and *RowHeights* properties to the values saved with the grid.

```
procedure DefineProperties(Filer: TFiler); override;
```

Description *DefineProperties* overrides the inherited method to save the *ColWidths* and *RowHeights* properties with the grid when they contain values other than the default widths and heights.

See also *ColWidths* property, *DefaultColWidth* property, *DefaultRowHeight* property, *RowHeights* property, *TFiler* object

DeleteColumn

DeleteColumn deletes the column with index *ACol* from the grid.

```
procedure DeleteColumn(ACol: Longint);
```

Description Call *DeleteColumn* to remove a column from the middle of the grid. To remove the last column from the grid, set the *ColCount* property.

See also *ColCount* property

DeleteRow

DeleteRow deletes the row with index *ARow* from the grid.

```
procedure DeleteRow(ARow: Longint);
```

Description Call *DeleteRow* to remove a row from the middle of the grid. To remove the last row from the grid, set the *RowCount* property.

See also *RowCount* property

Destroy

Destroy destroys an instance of *TCustomGrid*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the grid has not already been freed, and only then calls *Destroy*.

See also *Free* method

DoExit

DoExit hides the inplace editor when selection moves to another control in a form.

```
procedure DoExit; override;
```

Description After calling the inherited method, which generates an *OnExit* event, *DoExit* hides the inplace editor unless the *Options* property includes *goAlwaysShowEditor*.

See also *HideEditor* method, *Options* property, *TWinControl.OnExit* event

DrawCell

DrawCell is the abstract prototype for a method that draws the contents of a cell in the grid.

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);

procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState); virtual;
  abstract;
```

Description Descendants of *TCustomGrid* implement the *DrawCell* method to draw the contents of the cell or to generate an *OnDrawCell* event. If the *DefaultDrawing* property is *True*, the background of the cell is painted before *DrawCell* is called, and the 3D effects of the fixed cells or the focus rectangle around the focused cell are drawn after *DrawCell*.

The *ACol* and *ARow* parameters give the column and row indexes of the cell that should be drawn. The *ARect* parameter gives the boundaries of the cell in the coordinates of the canvas. The *AState* parameter indicates whether the cell is selected, whether the cell has input focus, and whether the cell is in the fixed, non-scrolling portion of the grid.

See also *DefaultDrawing* property, *FixedCols* property, *FixedRows* property, *Options* property

GetEditLimit

GetEditLimit provides the interface for a method that limits the number of characters that can be typed into cells.

```
function GetEditLimit: Integer; dynamic;
```

Description *GetEditLimit* is called by the inplace editor to determine if a limit should be placed on the number of characters a user can type into a cell. *GetEditLimit* sets no limits. Descendants of *TCustomGrid* override this method to limit the text for certain cells.

See also *InplaceEditor* property

GetEditMask

GetEditMask provides the interface for a method that returns an input mask for the specified cell.

```
function GetEditMask(ACol, ARow: Longint): string; dynamic;
```

Description The in-place editor calls *GetEditMask* to obtain any input mask associated with the cell specified by the *ACol* and *ARow* parameters. *GetEditMask* returns an empty string, indicating that the cell has no input mask. Descendants of *TCustomGrid* override this method to provide an input mask or generate an *OnGetEditMask* event.

See also *InplaceEditor* property, *TCustomMaskEdit.EditMask* property

GetEditText

GetEditText returns the value of the indicated cell formatted for editing.

```
function GetEditText(ACol, ARow: Longint): string; dynamic;
```

Description The inplace editor calls *GetEditText* to obtain the text for the indicated cell. *GetEditText* returns an empty string. Descendants of *TCustomGrid* override the *GetEditText* method to return a string that represents the contents of the cell indicated by the *ACol* and *ARow* parameters.

See also *InplaceEditor* property

GetGridHeight

GetGridHeight is the protected property access method for the *GridHeight* property.

```
function GetGridHeight: Integer;
```

Description *GetGridHeight* computes the value of the *GridHeight* property, based on the row heights, separator width, and the size of the control.

See also *ClientHeight* property, *GetGridWidth* method, *GridHeight* property, *GridLineWidth* property, *RowHeights* property

GetGridWidth

GetGridWidth is the protected property access method for the *GridWidth* property.

```
function GetGridWidth: Integer;
```

Description *GetGridWidth* computes the value of the *GridWidth* property, based on the column widths, separator width, and the size of the control.

See also *ClientWidth* property, *ColWidths* property, *GetGridHeight* method, *GridWidth* property, *GridLineWidth* property

HideEditor

HideEditor hides the inplace editor when it is not in use.

```
procedure HideEditor;
```

Description Grid controls call *HideEditor* when the grid goes out of edit mode. *HideEditor* hides the window of the inplace editor.

See also *EditorMode* property, *InplaceEditor* property, *ShowEditor* method

InvalidateCell

InvalidateCell invalidates the region occupied by a cell so that it will be repainted with the next Windows paint message.

```
procedure InvalidateCell(ACol, ARow: Longint);
```

Description Call *InvalidateCell* when the contents of a cell change so that the cell will need to be repainted. The *ACol* parameter is the column index of the cell, and the *ARow* parameter is the row index.

See also *CellRect* method, *InvalidateCol* method, *InvalidateEditor* method, *InvalidateRow* method

InvalidateCol

InvalidateCol invalidates the region occupied by a column so that it will be repainted with the next Windows paint message.

```
procedure InvalidateCol(ACol: Longint);
```

Description Call *InvalidateCol* when a column changes so that the area it occupies will need to be repainted. The *ACol* parameter is the column index.

See also *BoxRect* method, *InvalidateCell* method, *InvalidateRow* method

InvalidateEditor

InvalidateEditor causes the inplace editor to repaint itself, moving to a new location if necessary.

```
procedure InvalidateEditor;
```

Description *InvalidateEditor* is called when the selected cell changes while the grid is in edit mode.

See also *EditorMode* property, *HideEditor* method, *InplaceEditor* property, *InvalidateCell* method, *ShowEditor* method

InvalidateRow

InvalidateRow invalidates the region occupied by a row so that it will be repainted with the next Windows paint message.

```
procedure InvalidateRow(ARow: Longint);
```

Description Call *InvalidateRow* when a row changes so that the area it occupies will need to be repainted. The *ARow* parameter is the row index.

See also *BoxRect* method, *InvalidateCell* method, *InvalidateCol* method

KeyDown

The *KeyDown* provides special processing when the user presses a key.

```
type TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

procedure KeyDown(var Key: Word; Shift: TShiftState); override;
```

Description After generating an *OnKeyDown* event, *KeyDown* calls *CanGridAcceptKey* to screen out problematic key combinations. *KeyDown* moves to the appropriate cell in response to navigational key combinations, or sets the *EditorMode* property to *True* when the user presses *F2*.

See also *CanGridAcceptKey* method, *EditorMode* property, *KeyPress* method, *TWinControl.OnKeyDown* event

KeyPress

KeyPress provides special processing when the user presses a key.

```
procedure KeyPress(var Key: Char); override;
```

Description After generating an *OnKeyPress* event, *KeyPress* toggles the *EditorMode* property when the user presses the *Enter* key. *KeyPress* is called after the *KeyDown* method, but before the user releases the key.

See also *EditorMode* property, *KeyDown* method, *TWinControl.OnKeyPress* event

MouseCoord

MouseCoord returns the row and column indexes of the cell that contains a point specified in screen coordinates.

```
TGridCoord = record
  X: Longint;
  Y: Longint;
end;

function MouseCoord(X, Y: Integer): TGridCoord;
```

Description Call *MouseCoord* to locate the column and row of the cell which contains a given screen coordinate. Usually, *MouseCoord* is used to locate the cell which is under the mouse.

See also *CellRect* method, *MouseDown* method, *MouseMove* method, *MouseUp* method

MouseDown

MouseDown provides special processing when the user presses a mouse button.

```
procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); override;
```

Description *MouseDown* hides the editor window so that it can move selection to the cell that receives the mouse click. If the mouse lands in the hot region for beginning a move or resize of a row or column, *MouseDown* starts the move or resize, and provides visual feedback by changing the cursor. If the editor window was hidden, it is brought back up in its new location. Finally, *MouseDown* generates an *OnMouseDown* event.

See also *HideEditor* method, *InplaceEditor* property, *MouseMove* method, *MouseUp* method, *ShowEditor* method, *TControl.OnMouseDown* event

C

MouseMove

MouseMove provides special processing when the user moves the mouse.

```
procedure MouseMove(Shift: TShiftState; X, Y: Integer); override;
```

Description If the grid is in the middle of moving or resizing a row or column, or in the middle of selecting a range of cells using the mouse, *MouseMove* calculates the changes implied by the mouse move and provides visual feedback to the user. Finally, *MouseMove* generates an *OnMouseMove* event.

See also *MouseDown* method, *MouseUp* method, *TControl.OnMouseMove* event

MouseUp

MouseUp provides special processing when the user releases the mouse button.

```
procedure MouseUp(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); override;
```

Description *MouseUp* finishes a move or resize operation on a row or column, or if the user is selecting a range of cells using the mouse, *MouseUp* finishes the select operation. Finally, *MouseUp* generates an *OnMouseUp* event.

See also *MouseDown* method, *MouseMove* method, *MoveColumn* method, *MoveRow* method, *TControl.OnMouseUp* event

MoveColRow

MoveColRow selects the cell indicated by the *ACol* and *ARow* parameters.

```
procedure MoveColRow(ACol, ARow: Longint; MoveAnchor, Show: Boolean);
```

Description Grid controls call *MoveColRow* internally when changing the selected cell or cells. *MoveColRow* selects the indicated cell, but does not take care of removing any previous selection.

The *ACol* and *ARow* parameters are the column index and the row index of the cell to be selected. The *MoveAnchor* parameter indicates whether the new cell becomes the upper left corner of the selected region. *MoveAnchor* is *True* unless *MoveColRow* is called when extending a multicell selected region. The *Show* parameter indicates whether *MoveColRow* should ensure that the newly selected cell is visible.

See also *Col* property, *Row* property, *SelectCell* method, *Selection* property

MoveColumn

MoveColumn moves a column to a new position.

```
procedure MoveColumn(FromIndex, ToIndex: Longint);
```

Description Call *MoveColumn* to move the column identified by the *FromIndex* parameter to the position specified by the *ToIndex* parameter.

See also *ColumnMoved* method, *ColWidthsChanged* method

MoveRow

MoveRow moves a row to a new position.

```
procedure MoveRow(FromIndex, ToIndex: Longint);
```

Description Call *MoveRow* to move the row identified by the *FromIndex* parameter to the position specified by the *ToIndex* parameter.

See also *RowHeightsChanged* method, *RowMoved* method

Paint

Paint draws the image of the grid control on the screen.

```
procedure Paint; override;
```

Description *Paint* is called when the grid receives a Windows paint message. *Paint* draws the grid, calling the *DrawCell* method when it is time to paint the contents of each cell. If *DefaultDrawing* is *False*, *Paint* does not draw the background of the cells, provide the raised 3D effects on the fixed cells, or draw the focus rectangle around the selected cell when it has focus.

See also *DefaultDrawing* property, *DrawCell* method, *InvalidateCell* method, *InvalidateCol* method, *InvalidateRow* method

RowHeightsChanged

RowHeightsChanged responds when the row heights change.

```
procedure RowHeightsChanged; dynamic;
```

Description *RowHeightsChanged* is called immediately after the row heights change. The change can result from setting the *RowHeights* property, from setting the *DefaultRowHeight* property, from moving one of the rows, or from resizing the rows with the mouse.

RowHeightsChanged adjusts the scroll bar and the position of the inplace editor to take into account the new row size and positions of the rows.

See also *DefaultRowHeight* property, *InplaceEditor* property, *RowHeights* property, *ScrollBars* property

RowMoved

RowMoved provides the interface for a method that responds when the position of a row changes.

```
procedure RowMoved(FromIndex, ToIndex: Longint); dynamic;
```

Description *RowMoved* is called immediately after a row in the grid changes position. The *FromIndex* parameter is the old index of the row, and the *ToIndex* parameter is the new index. The *RowMoved* method of *TCustomGrid* does nothing. Descendants of *TCustomGrid* override this method to make internal adjustments or to generate an *OnRowMoved* event.

See also *MoveRow* method

ScrollData

ScrollData scrolls the visual image of the cells in the grid.

```
procedure ScrollData(DX, DY: Integer);
```

Description Grid controls call *ScrollData* internally when the nonfixed rows or columns scroll. *ScrollData* updates the image of the grid.

See also *LeftCol* property, *TopRow* property

SelectCell

SelectCell determines whether a particular cell in the grid can be selected.

```
function SelectCell(ACol, ARow: Longint): Boolean; virtual;
```

Description Grid controls call *SelectCell* internally before attempting to move selection to a the cell identified by the *ACol* and *ARow* parameters. *ACol* and *ARow* are the column and row indexes for the cell, where the first column has index 0, and the top row has index 0.

The *SelectCell* method for *TCustomGrid* always returns *True*, allowing the cell to be selected. Descendants of *TCustomGrid* override this method to check the properties of the information displayed in the cell or generating an *OnSelectCell* event.

See also *MoveColRow* method, *TabStops* property

SetEditText

SetEditText provides the interface for a method that updates the text associated with a cell.

```
procedure SetEditText(ACol, ARow: Longint; const Value: string); dynamic;
```

Description *SetEditText* is called by the inplace editor so that the grid can store the new value for the current cell. The *SetEditText* method for *TCustomGrid* does nothing. Descendants of *TCustomGrid* override this method to update an internal representation of the cell's value, or to generate an *OnSetEditText* event.

See also *GetEditText* method

ShowEditor

ShowEditor displays the inplace editor when the grid enters edit mode.

```
procedure ShowEditor;
```

Description Grid controls call *ShowEditor* when the grid enters edit mode. *ShowEditor* displays the window of the inplace editor over the currently selected cell. If the inplace editor does not yet exist, *ShowEditor* creates it.

See also *CreateEditor* method, *EditorMode* property, *HideEditor* method, *InplaceEditor* property

ShowEditorChar

ShowEditorChar displays the inplace editor, with its text set to the specified character.

```
procedure ShowEditorChar(Ch: Char);
```

Description Grid controls call *ShowEditorChar* when the user types a character into a cell and the editor is not already showing. *ShowEditorChar* displays the window of the inplace editor over the currently selected cell, and sets its text property to the character *Ch*. If the inplace editor does not yet exist, *ShowEditorChar* creates it.

See also *CreateEditor* method, *EditorMode* property, *HideEditor* method, *InplaceEditor* property, *ShowEditor* method

SizeChanged

SizeChanged responds to changes in the size of the grid.

```
procedure SizeChanged(OldColCount, OldRowCount: Longint); dynamic;
```

Description *SizeChanged* provides the interface for a method that responds to changes in the number of rows or columns in the grid. The *SizeChanged* method of *TCustomGrid* does nothing. Descendants of *TCustomGrid* override this method to make internal adjustments or generate events.

See also *ColCount* property, *RowCount* property

Sizing

Sizing indicates whether the indicated coordinates are on one of the resize hot zones of the graph.

```
function Sizing(X, Y: Integer): Boolean;
```

Description Grid controls call *Sizing* internally with the coordinates of a mouse click to determine whether the user is about to resize a row or column.

See also *MouseDown* method

C

TimedScroll

TimedScroll extends the move, resize, or selection out of the scrollable region when the user drags the mouse in a direction where the grid is scrolled.

```
type TGridScrollDirection = set of (sdLeft, sdRight, sdUp, sdDown);

procedure TimedScroll(Direction: TGridScrollDirection); dynamic;
```

Description *TimedScroll* is called periodically as the user continues to drag with the mouse outside of the scrollable cells in the grid. *TimedScroll* does nothing unless the user is moving a row or column, resizing a row or column, or selecting a multiple cell region using the mouse. *TimedScroll* scrolls the cells in the grid to allow the operation to continue into the portion of the grid that was hidden due to scrolling.

See also *LeftCol* property, *MouseMove* method, *TopRow* property, *VisibleColCount* property, *VisibleRowCount* property

TopLeftChanged

TopLeftChanged updates the position of the inplace editor when the grid scrolls.

```
procedure TopLeftChanged; dynamic;
```

Description Grid controls call *TopLeftChanged* internally, whenever the *LeftCol* or *TopRow* property changes.

Descendants of *TCustomGrid* override this method to make additional adjustments or to generate an *OnTopLeftChanged* event.

See also *InplaceEditor* property, *LeftCol* property, *TopRow* property

UpdateDesigner

UpdateDesigner notifies the designer when rows and columns of the grid are moved or resized.

```
procedure UpdateDesigner;
```

Description At design time, the designer automatically keeps track of property changes so that it can prompt the user to save any changes. Moving or resizing the rows and columns of the grid using the mouse bypasses this system. When such changes occur, *UpdateDesigner* notifies the designer that the properties of the grid have changed and that the changes should be saved.

See also *MouseUp* method

TCustomGroupBox

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomGroupBox

TCustomGroupBox is the abstract base class for all group-box components.

Description Group boxes are visual components that act as containers for sets of related controls, such as radio buttons. Group boxes may also contain other group boxes. *TCustomGroupBox* is the abstract class from which all group-box components—including *TGroupBox*, *TRadioGroup*, and *TDBRadioGroup*—descend.

In addition to the methods it inherits from *TCustomControl*, *TCustomGroupBox* overrides the *AlignControls* method (inherited from *TWinControl*) to handle positioning of components within the group box. *TCustomGroupBox* also overrides the *Paint* and *Create* methods.

See also *TCustomRadioGroup* object, *TDBRadioGroup* object, *TRadioGroup* object, *TGroupBox* object, *TCustomControl* object, *TWinControl* object

Properties

The following table lists the properties of *TCustomGroupBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl

Property	Visibility	Default	Documented in
Width	published		TControl
WindowProc	public		TControl

Methods

The following table lists the methods of *TCustomGroupBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomGroupBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCustomControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent

Method	Visibility	Documented in
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Paint	protected	TCustomGroupBox
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Creates and initializes a *TCustomGroupBox* instance.

constructor Create(AOwner: TComponent); **override**;

Description After calling the inherited constructor method, *Create* initializes the *ControlStyle* property by turning on the following flags: *csAcceptsControls*, *csCaptureMouse*, *csClickEvents*, *csSetCaption*, *csDoubleClicks*, and *csReplicatable*. *Create* also sets the *Width* and *Height* properties to 185 and 105 pixels, respectively.

See also *TControl* object, *ControlStyle* property, *TControlStyle* type, *Width* property, *Height* property, *TCustomControl.Create* method

Paint

Renders the onscreen image of the group box.

procedure Paint; **override**;

Description *Paint* creates a Windows *RECT* structure, draws a border around it, and calls the Windows *DrawText* function to fill in the caption. If *Ctl3D* (a protected property, inherited from *TWinControl*) is set to *True*, then *Paint* draws the border with a three-dimensional look.

See also *TWinControl.Ctl3D* property

TCustomHotKey

Unit: ComCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomHotKey

TCustomHotKey is the abstract base class type for the hot key object that is a wrapper for the Windows hot key Common Control.

Description *TCustomHotKey* is an abstract class that encapsulates behavior of an object that is used to set a shortcut property at runtime. Hot key objects are used to perform an action quickly by enabling users to link the hot key to a combination of keystrokes that will execute that action. *TCustomHotKey* introduces properties to:

- Associate the hot key with keystrokes to perform an action
- Select a key combination at design-time
- Invalidate specific modifier keys

To change the key combination at runtime hold down the *Alt*, *Ctrl* or *Shift* keys or combinations of them and type a letter or number. The hot key property can then be assigned to the shortcut property of a menu item, for example.

Hot key classes that descend from *TCustomHotKey* inherit the basic functionality of *TCustomHotKey*. For component writers wanting to create a customized hot key objects, use *TCustomHotKey* as a base class for deriving specialized hot key objects.

Properties

The following table lists the properties of *TCustomHotKey*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
AutoSize	protected	True	TCustomHotKey
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
HotKey	protected		TCustomHotKey
InvalidKeys	protected		TCustomHotKey
Left	published		TControl
Modifiers	protected		TCustomHotKey
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl

Property	Visibility	Default	Documented in
TabOrder	public	–1	TWinControl
TabStop	protected	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
Visible	public	True	TControl
Width	published		TControl

AutoSize

AutoSize determines whether the hot key automatically changes its height to accommodate the font size.

```
property AutoSize: Boolean;
```

Description Use *AutoSize* to fully display the font on the hot key control. When *AutoSize* is *True*, the height of the hot key changes to display the key combinations typed in it. When *AutoSize* is *False*, the hot key remains the same height, regardless of any font changes. The default value is *True*.

HotKey

HotKey contains the current key combination of the hot key control.

```
type TShortCut = Low(Word) .. High(Word);

property HotKey: TShortCut;
```

Description Use *HotKey* to set or change the key combination associated with the hot key. *Alt+A* is the default. To modify the value of *HotKey* either edit it directly or set the value of the *Modifiers* property. At runtime *HotKey* is reset when the hot key has focus and the user presses valid key combinations.

See also *Modifiers* property

InvalidKeys

InvalidKeys allows the user to specify that one or more modifier keys should be considered invalid.

```
type
  THKInvalidKey = (hcNone, hcShift, hcCtrl, hcAlt, hcShiftCtrl, hcShiftAlt, hcCtrlAlt, hcShiftCtrlAlt);
  THKInvalidKeys = set of THKInvalidKey;

property InvalidKeys: THKInvalidKeys;
```

Description Use *InvalidKeys* to select which modifier keys are not allowed. If this property is set, a default modifier should also be specified in the *Modifiers* property. When the invalid key is selected, the default modifier is displayed in its place.

THKInvalidKey is the type of the *InvalidKeys* property. These are the possible values:

Value	Meaning
hcNone	Unmodified keys are invalid.
hcShift	The *Shift* key is invalid as a modifier.
hcCtrl	The *Ctrl* key is invalid as a modifier.
hcAlt	The *Alt* key is invalid as a modifier.
hcShiftCtrl	The *Shift+Ctrl* key combination is invalid as a modifier.
hcShiftAlt	The *Shift+Alt* key combinations invalid as a modifier.
hcCtrlAlt	The *Ctrl+Alt* key combination is invalid as a modifier.
hcShiftCtrlAlt	The *Shift+Ctrl+Alt* key combination is invalid as a modifier.

The default values are *hcNone* and *hcShift*.

See also *Modifiers* property

Modifiers

Modifiers specifies the key or keys that are used in combination with a non-modifier key.

```
type
  THKModifier = (hkShift, hkCtrl, hkAlt, hkExt);
  THKModifiers = set of THKModifier;

property Modifiers: THKModifiers;
```

Description Use *Modifiers* to select a modifier key such as Ctrl, Alt, or Shift for the hot key. These are the possible values:

Value	Meaning
hkShift	The *Shift* key is used as a modifier.
hkCtrl	The *Ctrl* key is used as a modifier.
hkAlt	The *Alt* key is used as a modifier.
hkExt	The *Extra* key is used as a modifier.

The default value is *hkAlt*.

Modifier keys are used in combination with another non-modifier key such as character and function keys, arrow keys, and so on. More than one modifier key can be used.

Methods

The following table lists the methods of *TCustomHotKey*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomHotKey
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent

Method	Visibility	Documented in
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

C

Create

Create instantiates a hot key control.

Description Call *Create* to instantiate a hot key control at runtime. For hot key controls created at design time, *Create* is called automatically.

Create allocates memory for a hot key control and calls the inherited *Create*. Then it sets default values for the *Width* and *Height* properties, and initializes the following properties:

- *TabStop* to *True*
- *ParentColor* to *False*
- *AutoSize* to *True*
- *InvalidKeys* to *hcNone* and *hcShift*
- *Modifiers* to [*hkAlt*]
- *HotKey* to *'Alt+A'*

See also *TControl.ParentColor* method, *TWinControl.Create* method

TCustomImageList

Unit: Controls

TObject • TPersistent • TComponent • TCustomImageList

A *TCustomImageList* is an abstract class that is derived from to provide a collection of same-sized images, each of which can be referred to by its index.

Description Image lists are used to efficiently manage large sets of icons or bitmaps. All images in an image list are contained in a single, wide bitmap in screen device format. An image list may also include a monochrome bitmap that contains masks used to draw images transparently (icon style).

See also *TListView* object, *TTreeView* object

Properties

The following table lists the properties of *TCustomImageList*:

Property	Visibility	Default	Documented in
AllocBy	protected	4	TCustomImageList
BkColor	protected	clNone	TCustomImageList
BlendColor	protected	clNone	TCustomImageList
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Count	public, read-only		TCustomImageList
DesignInfo	public		TComponent
DragCursor	public		TCustomImageList
Dragging	public, read-only		TCustomImageList
DrawingStyle	protected	dsNormal	TCustomImageList
Handle	public		TCustomImageList
Height	protected	16	TCustomImageList
ImageType	protected	itImage	TCustomImageList
Masked	protected	True	TCustomImageList
Name	published		TComponent
Owner	public, read-only		TComponent
ShareImages	protected	False	TCustomImageList

Property	Visibility	Default	Documented in
Tag	published	0	TComponent
VCLComObject	public		TComponent
Width	protected	16	TCustomImageList

AllocBy

The *AllocBy* property sets the number of images which the image list will grow by when it needs to resize to make room for new images.

```
property AllocBy: Integer;
```

Description Use the *AllocBy* method to specify the number of new images that can be added to the resized image list.

See also *Count* property, *Add* method, *AddIcon* method, *AddImages* method, *AddMasked* method

BkColor

The *BkColor* property determines which background color to use when drawing an image.

```
property BkColor: TColor;
```

Description Setting BKColor to *clNone* specifies no background color i.e. the image is drawn transparently. Setting BKColor to *clDefault* specifies using the image lists background color when drawing.

See also *BlendColor* property

C

BlendColor

The *BlendColor* property determines which foreground color to use when drawing an image.

```
property BlendColor: TColor;
```

Description *clNone* specifies no blend color while *clDefault* specifies the system highlight color for the foreground color.

See also *BkColor* property

Count

The *Count* property returns the number of images in the image list.

```
property Count: Integer;
```

Description *Count* can be useful when iterating through the image list.

See also *AllocBy* property

DragCursor

The *DragCursor* property specifies the current drag cursor that is associated with the image list.

```
property DragCursor: TCursor;
```

Description Use the *DragCursor* property to determine which drag cursor should be displayed when an image in the image list is dragged. When an image is dragged, a new drag image is created by combining the given image (typically a mouse cursor image) with the current drag cursor.

See also *BeginDrag* method, *Dragging* property

Dragging

The *Dragging* property returns whether an image in the image list is being dragged.

```
property Dragging: Boolean;
```

Description When *Dragging* returns *True*, an image in the image list is being dragged. When *Dragging* returns *False*, no image is being dragged.

See also *BeginDrag* method

DrawingStyle

The *DrawingStyle* property specifies the style to be used when the image list is drawing an image.

```
type TDrawingStyle = (dsFocus, dsSelected, dsNormal, dsTransparent);
property DrawingStyle: TDrawingStyle;
```

Description Use the *DrawingStyle* property to control how the image is to appear. These are the possible values:

Value	Meaning
dsFocused	Draws the image blending 25% with the system highlight color. This only affects image lists which contain masks.
dsSelected	Draws the image blending 50% with the system highlight color. This only affects image lists which contain masks.
dsNormal	Draws the image using the color specified in the *BkColor* property. If the *BkColor* is *clNone* then the image is drawn transparently using the mask.
dsTransparent	Draws using the mask regardless of the *BkColor* setting.

See also *Draw* method, *DrawOverlay* method, *ImageType* property

Handle

The *Handle* property specifies the handle to the image list.

property Handle;

Description The *Handle* property allows access to the Windows image list object handle when calling Windows API functions that requires it.

See also *HandleAllocated* method

Height

The *Height* property specifies the height of images within the image list.

property Height: Integer;

Description Use the *Height* property to specify the height of images before any are added to the image list. The image list is cleared when this value changes.

See also *Width* property, *CreateSize* method

ImageType

The *ImageType* property determines whether the image list will use the image or the associated image mask when drawing.

property ImageType: TImageType;

Description Use the *ImageType* property to control how the image is to appear. These are the possible values

Value	Meaning
itImage	Draws the image.
itMask	Draws the mask.

See also *DrawingStyle* property, *Draw* method, *Masked* property, *DrawOverlay* property

Masked

The *Masked* property specifies whether the image list contains transparent or non-transparent images.

property Masked: Boolean;

Description Set the *Masked* property to *True* to specify that the images will contain transparent areas.

See also *AddMasked* method, *InsertMasked* method

ShareImages

The *ShareImages* property specifies whether the image list destroys its handle when the image list is destroyed.

property ShareImages: Boolean;

Description Set the *ShareImages* property to *False* to ensure that the handle to the image list is freed when the image list is destroyed.

See also *Destroy* method

Width

The *Width* property sets the width of images within the image list.

property Width: Integer;

Description Use the *Width* property to specify the width of images before any are added to the image list. The image list is cleared when this value changes.

See also *Height* property, *CreateSize* method

Events

The following table lists the events of *TCustomImageList*:

Event	Visibility	Documented in
OnChange	protected	TCustomImageList

On Change

The *OnChange* event is called when the image list changes.

property OnChange: TNotifyEvent;

Description Use the *OnChange* event to execute some code when the image list changes. Any objects registered with the *RegisterChanges* method will also be notified.

See also *RegisterChanges* method

Methods

The following table lists the methods of *TCustomImageList:*

Method	Visibility	Documented in
Add	public	TCustomImageList
AddIcon	public	TCustomImageList
AddImages	public	TCustomImageList

Method	Visibility	Documented in
AddMasked	public	TCustomImageList
Assign	public	TCustomImageList
BeginDrag	public	TCustomImageList
Change	protected	TCustomImageList
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomImageList
Create	public	TCustomImageList
CreateSize	public	TCustomImageList
DefaultHandler	public	TObject
DefineProperties	protected	TCustomImageList
Delete	public	TCustomImageList
Destroy	public	TCustomImageList
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
DragLock	public	TCustomImageList
DragMove	public	TCustomImageList
DragUnlock	public	TCustomImageList
Draw	public	TCustomImageList
DrawOverlay	public	TCustomImageList
EndDrag	public	TCustomImageList
FieldAddress	public	TObject
FileLoad	public	TCustomImageList
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetBitmap	public	TCustomImageList
GetHotSpot	public	TCustomImageList
GetIcon	public	TCustomImageList
GetImageBitmap	public	TCustomImageList
GetImages	protected	TCustomImageList
GetInstRes	public	TCustomImageList
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject

Method	Visibility	Documented in
GetMaskBitmap	public	TCustomImageList
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetResource	public	TCustomImageList
HandleAllocated	public	TCustomImageList
HandleNeeded	protected	TCustomImageList
HasParent	public	TComponent
HideDragImage	public	TCustomImageList
InheritsFrom	public	TObject
Initialize	protected	TCustomImageList
InitInstance	public	TObject
Insert	public	TCustomImageList
InsertComponent	public	TComponent
InsertIcon	public	TCustomImageList
InsertMasked	public	TCustomImageList
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
Move	public	TCustomImageList
NewInstance	public	TObject
Overlay	public	TCustomImageList
RegisterChanges	public	TCustomImageList
RemoveComponent	public	TComponent
Replace	public	TCustomImageList
ReplaceIcon	public	TCustomImageList
ReplaceMasked	public	TCustomImageList
ResInstLoad	public	TCustomImageList
ResourceLoad	public	TCustomImageList
SafeCallException	public	TComponent
SetDragImage	public	TCustomImageList
ShowDragImage	public	TCustomImageList
UnRegisterChanges	public	TCustomImageList

Add

The *Add* method adds an *Image* and its *Mask* to the image list object. *Add* returns the index of the added image.

```
function Add(Image, Mask: TBitmap): Integer;
```

Description Use the *Add* method to add images with their masks to an image list. When the *Masked* property of the image list is *False*, the second bitmap handle is ignored; it can be set to *nil*. *Add* returns –1 if it fails, 0 if it's successful.

See also *AddIcon* method, *AddImages* method, *AddMasked* method

AddIcon

The *AddIcon* method adds an icon specified by *Image* to the image list object. *AddIcon* returns the index of the added image.

```
function AddIcon(Image: TIcon): Integer;
```

Description Use the *AddIcon* method to add an icon to an image list. *AddIcon* returns –1 if it fails, 0 if it's successful.

See also *Add* method, *AddImages* method, *AddMasked* method

AddImages

The *AddImages* method adds images from another image list specified by *Value* to the image list object.

```
procedure AddImages(Value: TCustomImageList);
```

Description Use the *AddImages* method to add the images from another image list. *AddImages* returns –1 if it fails, 0 if it's successful.

See also *Add* method, *AddIcon* method, *AddMasked* method

AddMasked

The *AddMasked* method adds an image to the image list along with a color which is used to generate the transparent mask from the image.

```
function AddMasked(Image: TBitmap; MaskColor: TColor): Integer;
```

Description *AddMasked* is similar to *Add* except that a mask bitmap is not specified. Instead, a color that the system combines with the image bitmap to automatically generate the masks is specified. Each pixel of the specified color in the image bitmap is changed to black, and the corresponding bit in the mask is set to one. As a result, any pixel in the image that matches the specified color is transparent when the image is drawn. *AddMasked* returns –1 if it fails, 0 if it's successful.

See also *Add* method, *AddIcon* method, *AddImages* method

Assign

The Assign method copies the information from one image list to another.

```
procedure Assign(Source: TPersistent);
```

Description If *Source* is another *TImageList* object, *Assign* discards any current image information and replaces it with the information from *Source*. If *Source* is any

other type of object, *Assign* calls its inherited method. Use this method to copy one object's information to another.

See also *AddImages* method, *Add* method

BeginDrag

The *BeginDrag* method starts the dragging of an image.

```
function BeginDrag(Window: HWND; X, Y: Integer): Boolean;
```

Description Use the *BeginDrag* method to start dragging an image set by *SetDragImage* at the *X* and *Y* coordinates of the drag position relative to the upper-left corner of the image. *Window* is the handle to the image list. *BeginDrag* returns *True* if it was successful in starting the drag of the image.

See also *EndDrag* method, *DragLock* method

Change

The *Change* method notifies the image list and any objects registered by *RegisterChanges* that a change has occurred in the image list.

```
procedure change;
```

Description Use this method to trigger the *OnChange* event of the image list and any objects registered with the image list for change notification.

See also *RegisterChanges* method, *OnChange* event

Clear

The *Clear* method clears all images from the image list.

```
procedure Clear;
```

Description Use the *Clear* method to remove all images from the image list. Use the *Delete* method to delete one image from the image list.

See also *Delete* method

Create

The *Create* constructor creates an empty image list.

```
constructor Create(AOwner: TComponent); override;
```

Description Use the *Create* constructor to create an image list as a child of the component specified by *AOwner* with a default image size of 16 by 16 pixels.

See also *CreateSize* method

CreateSize

The *CreateSize* constructor creates a *TImageList* object of a specified size.

constructor CreateSize(AWidth, AHeight: Integer);

Description Use the *CreateSize* constructor to create an image list object with an image size that other than the default of 16 x 16 pixels. The initial size of the image list is determined by the *AHeight* and *AWidth* parameters. The height and width can be altered after the image list has been created using the *Width* and *Height* properties.

See also *Create* method, *Height* property, *Width* property

DefineProperties

The *DefineProperties* method calls the inherited *DefineProperties* method and also defines bitmap data for storage by the filer.

procedure DefineProperties(Filer: TFiler);

Description The *DefineProperties* method calls the inherited *DefineProperties* method and also calls the *TFiler DefineBinaryProperty* which defines binary data the filer object will store as if the data were a property.

See also *TControl.DefineProperties* method, *TFiler.DefineBinaryProperty* method

Delete

The *Delete* method deletes a particular image from the image list.

procedure Delete(Index: Integer);

Description Use the *Delete* method to delete an image specified by *Index* in the image list. Use the *Clear* method to delete all images in the image list.

See also *Clear* method

Destroy

The *Destroy* method destroys the image list and releases the memory allocated to it.

destructor Destroy;

Description Before calling *Free*, *Destroy* first calls *UnRegisterChanges* to unregister any objects registered to receive notification when the image list changed and frees all *TChangeLink* and *TBitMap* objects associated with the image list.

Destroy seldom needs to be called. Objects designed with Delphi create and destroy themselves as needed. When an image list is constructed by calling the *Create* or *CreateSize* method, *Free* should be called to release memory and

dispose of the object. *Free* checks to see if the pointer is *nil* before calling *Destroy*.

See also *Free* method

DragLock

The *DragLock* method associates the drag image with the given Window handle and draws it at the position given by the parameters *XPos* and *YPos* while the window is in drag mode. Returns *True* if successful.

```
function DragLock(Window: HWND; XPos, YPos: Integer): Boolean;
```

Description Call *DragLock* to use an image in the image list as a drag image when a particular window is dragged.

See also *SetDragImage* method, *ShowDragImage* method

DragMove

The *DragMove* method moves the drag image set by *SetDragImage* to a new position specified by *X* and *Y* within its currently associated window while the window is in drag mode. Returns *True* if successful.

```
function DragMove(X, Y: Integer): Boolean;
```

Description Use the *DragMove* method to move the image that is being dragged during a drag-and-drop operation.

See also *BeginDrag* method, *Dragging* property

DragUnlock

The *DragUnlock* method removes the association between the drag image and a window.

```
procedure DragUnlock;
```

Description Use the *DragUnlock* method when an image is no longer needed as a drag image for a particular window.

See also *DragLock* method

Draw

The *Draw* method draws the image specified by the *Index* parameter onto the provided *Canvas* at position *X, Y*. The image is drawn according to the style set in the *DrawingStyle* property.

```
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer);
```

Description Use the *Draw* method to draw an image.

Example The following code draws each header section adding text for a title and icons stored in ImageList1. The image list contains as many icons as there are header sections.

```
procedure TForm1.HeaderControl1DrawSection(HeaderControl: THeaderControl;
  Section: THeaderSection; const Rect: TRect; Pressed: Boolean);
begin
  with HeaderControl1.Canvas do
  begin
    Brush.Color := clRed;
    FillRect(Rect);
    Font.Color := clYellow;
    TextOut(Rect.left + 30, Rect.top + 2, 'Header Control');
    ImageList1.Draw(HeaderControl1.Canvas,Rect.Left,Rect.Top,Section.Index);
  end;
end;
```

See also *DrawingStyle* property, *DrawOverlay* method

DrawOverlay

The *DrawOverlay* method draws an image and an overlay onto the provided *Canvas*.

procedure DrawOverlay(Canvas: TCanvas; X, Y: Integer; ImageIndex: Integer; Overlay: TOverlay);

Description An overlay mask is an image drawn transparently over another image. Any image can be used as an overlay mask. To draw an overlay mask over an image use the *DrawOverlay* method with the index of the overlay mask. The *Overlay* method must be called first to allow an image to be used as an overlay.

See also *Overlay* method, *Draw* method, *SetDragImage* method

C

EndDrag

The *EndDrag* method ends the drag operation. *EndDrag* returns *True* if successful.

function EndDrag: Boolean;

Description Use *EndDrag* after *BeginDrag* has been called to end the drag operation.

See also *BeginDrag* method, *OnEndDrag* event

FileLoad

The *FileLoad* method loads a file of type *TResType* into the image list. The *MaskColor* is used to set the transparent color for the images in the file. *Name* specifies the file to be loaded.

function FileLoad(ResType: TResType; Name: string; MaskColor: TColor): Boolean;

Description Use the *FileLoad* method to load a bitmap, cursor or icon file into an image list. *FileLoad* returns *True* if successful.

See also *GetResource* method, *ResourceLoad* method

GetBitmap

The *GetBitmap* method returns the image specified by the *Index* parameter as a bitmap into the *Image* parameter.

```
procedure GetBitmap(Index: Integer; Image: TBitmap);
```

Description Use the *GetBitmap* method to obtain a particular image in the image list as a bitmap object.

See also *GetMaskBitmap* method

GetHotSpot

The *GetHotSpot* method returns the hot spot location associated with the current drag image.

```
function GetHotSpot: TPoint;
```

Description Use the *GetHotSpot* method to obtain the position of the hot spot of the current drag image.

See also *DragMove* method, *SetDragImage* method

GetIcon

The *GetIcon* method returns the image specified by *Index* as an icon into the Image parameter.

```
procedure GetIcon(Index: Integer; Image: TIcon);
```

Description Use the *GetIcon* method to obtain a particular image in the image list as an icon.

See also *GetImageBitmap* method

GetInstRes

The *GetInstRes* method loads the specified bitmap, cursor, or icon resource into the image list.

```
function GetInstRes(Instance:THandle; ResType: TResType; Name: string; Width: Integer;
LoadFlags: TLoadResources; MaskColor: TColor):
```

Description Use the *GetInstRes* method to load bitmap, cursor, or icon resource of *Name* into an image list.

Note *GetInstRes* must be used to load a resource from within a package. Within a package, *GetResource* will return an error.

ResType specifies the type of graphical resource that is to be loaded into the image list and can be *rtBitmap*, *rtCursor* or *rtIcon*. The *MaskColor* is used to set the transparent color for the images in the file. *LoadFlags* specifies how to load the image. A combination of the following values can be used.

Value	Meaning
lrNone	No specific resource flags.
lrDefaultColor	Use the color format of the display.
lrDefaultSize	Use system metrics for *Height* and *Width* if the *Width* parameter to *GetResource* is 0. If the *Width* parameter is 0 and this option is not specified for cursors and icons, the resources width and height will be used.
lrFromFile	Loads the resource from the specified file.
lrMap3DColors	Replaces the images grey shades with the 3D windows colors.
lrTransparent	Replaces the color table entry for the first pixel of the resource with the default window color (COLOR_WINDOW). This applies only to images with color tables.
lrMonoChrome	Loads the resource as black and white.

See also *ResInstLoad* method, *ResourceLoad* method, *FileLoad* method, *GetResource* method

GetImageBitmap

The *GetImageBitmap* method returns a handle to a bitmap containing all of the images in the image list.

```
function GetImageBitmap: HBITMAP;
```

Description Use the *GetImageBitmap* method to obtain a bitmap that contains all images in the image list. *GetImageBitmap* returns 0 unsuccessful.

See also *GetImages* method, *GetBitmap* method

GetImages

The *GetImages* method returns the image and its mask specified by the *Index* parameter as a bitmap into the *Image* and *Mask* parameters respectively.

```
procedure GetImages(Index: Integer; Image, Mask: TBitmap);
```

Description Use the *GetImages* method to obtain an image and its mask as bitmaps.

See also *GetImageBitmap* method, *GetMaskBitmap* method, *GetBitmap* method

GetMaskBitmap

The *GetMaskBitmap* method returns a handle to a bitmap containing all masks for images in the image list.

```
function GetMaskBitmap: HBITMAP;
```

Description Use the *GetMaskBitmap* to obtain a bitmap that contains all masks in the image list. *GetMaskBitmap* returns 0 unsuccessful.

See also *GetImageBitmap* method, *GetImages* method

GetResource

The *GetResource* method loads the specified bitmap, cursor, or icon resource into the image list.

```
TLoadResource = (lrDefaultColor, lrDefaultSize, lrFromFile, lrMap3DColors, lrTransparent,
lrMonoChrome);

TLoadResources = Set of TLoadResource;

function GetResource(ResType: TResType; Name: string; Width: Integer; LoadFlags:
TLoadResources; MaskColor: TColor): Boolean;
```

Description Use the *GetResource* method to load bitmap, cursor, or icon resource of *Name* into an image list.

Note Within a package, *GetResouce* will return an error. *GetInstRes* must be used to load a resource from within a package.

ResType specifies the type of graphical resource that is to be loaded into the image list and can be *rtBitmap*, *rtCursor* or *rtIcon*. The *MaskColor* is used to set the transparent color for the images in the file. *LoadFlags* specifies how to load the image. A combination of the following values can be used.

Value	Meaning
lrNone	No specific resource flags.
lrDefaultColor	Use the color format of the display.
lrDefaultSize	Use system metrics for *Height* and *Width* if the *Width* parameter to *GetResource* is 0. If the *Width* parameter is 0 and this option is not specified for cursors and icons, the resources width and height will be used.
lrFromFile	Loads the resource from the specified file.
lrMap3DColors	Replaces the images grey shades with the 3D windows colors.
lrTransparent	Replaces the color table entry for the first pixel of the resource with the default window color (COLOR_WINDOW). This applies only to images with color tables.
lrMonoChrome	Loads the resource as black and white.

See also *ResourceLoad* method, *FileLoad* method, *GetInstRes* method

HandleAllocated

The *HandleAllocated* method returns *True* if a window handle for the control exists, *False* if none exists.

```
function HandleAllocated: Boolean;
```

Description Use the *HandleAllocated* method to determine if a handle exists for the control. *HandleAllocated* doesn't create a handle if one doesn't already exist. Calling the *Handle* property of a control directly creates a handle automatically if one didn't previously exist.

See also *TControl.Handle* property

HandleNeeded

The *HandleNeeded* method creates an image list handle for the image list if one doesn't already exist.

```
procedure HandleNeeded;
```

Description Use the *HandleNeeded* method to create an image list handle for the image list when one doesn't already exist. This method is called automatically when an image list's handle property is accessed.

See also *HandleAllocated* method

HideDragImage

The *HideDragImage* method hides the current drag image if it was previously shown.

```
procedure HideDragImage;
```

Description Use the *HideDragImage* method to hide the current drag image.

See also *ShowDragImage* method

Initialize

The *Initialize* method sets numerous image list properties to their default values and creates instances of objects used internal to the image list.

```
procedure Initialize;
```

Description *Initialize* is called from the *Create* and *CreateSize* constructors of the image list.

See also *Create* method, *CreateSize* method

Insert

The *Insert* method inserts a bitmap and its mask into the image list following the image specified by *Index*.

```
procedure Insert(Index: Integer; Image, Mask: TBitmap);
```

Description Use the *Insert* method to insert a bitmap and its mask into the image list.

See also *Add* method, *AddMasked* method

InsertIcon

The *InsertIcon* method inserts an icon at the *Index* + 1 position in the image list.

```
procedure InsertIcon(Index: Integer; Image: TIcon);
```

Description Use the *InsertIcon* method to insert an icon into the image list.

See also *AddIcon* method

InsertMasked

The *InsertMasked* method inserts a bitmap at the *Index* + 1 position, creating the mask from the transparent color provided by the parameter *MaskColor*.

```
procedure InsertMasked(Index: Integer; Image: TBitmap; MaskColor: TColor);
```

Description Use the *InsertMasked* method to insert a masked image into the image list.

See also *AddMasked* method

Move

The *Move* method moves an image specified by *CurIndex* in the image list to another position specified by *NewIndex* within the image list.

```
procedure Move(CurIndex, NewIndex: Integer);
```

Description Use the *Move* method to move an image to another position in the image list.

See also *Insert* method, *ImageIndex* property

Overlay

The *Overlay* method adds the index of an image to the list of images to be used as overlay masks. Up to four indices can be added to the list.

```
TOverlay = 0..3;

function Overlay(ImageIndex: Integer; Overlay: TOverlay): Boolean;
```

Description An overlay mask is an image drawn transparently over another image. Any image can be used as an overlay mask. To draw an overlay mask over an image use the *DrawOverlay* method with the index of the overlay mask. If successful, *Overlay* returns *True*.

See also *DrawOverlay* method

RegisterChanges

The *RegisterChanges* method registers an object to be notified when changes occur in the image list.

```
procedure RegisterChanges(Value: TChangeLink);
```

Description Use the *RegisterChanges* method to have an object notified whenever changes occur in the image list. The *TChangeLink*'s *OnChange* event is called whenever a change in the image list occurs.

Create a change link object for each object that should be notified of changes. Register each object by calling *RegisterChanges* and passing its change link object as the *Value* parameter. The *OnChange* event of all registered change link objects is then triggered when a change occurs in the image list. Any code assigned to the *OnChange* event handlers will execute.

See also *UnRegisterChanges* method, *TChangeLink* object, *TChangeLink.OnChange* event

Replace

The *Replace* method replaces an image in the image list with a new image and mask.

```
procedure Replace(Index: Integer; Image, Mask: TBitmap);
```

Description Use the *Replace* method to replace the image at the *Index* position in the image list with a new *Image* and overlay *Mask*.

See also *Insert* method, *ReplaceIcon* method, ReplaceMasked method

ReplaceIcon

The *ReplaceIcon* method replaces an image in the image list with an icon.

```
procedure ReplaceIcon(Index: Integer; Image: TIcon);
```

Description Use the *ReplaceIcon* method to replace the Image at the *Index* position in the image list with a new icon.

See also *Replace* method, *InsertIcon* method

ReplaceMasked

The *ReplaceMasked* method replaces an image in the image list with a new image and mask.

```
procedure ReplaceMasked(Index: Integer; NewImage: TBitmap; MaskColor: TColor);
```

Description Use the *ReplaceMasked* method to replace the image at *Index* position within the image list with a new image and mask created using the *MaskColor* for transparency.

See also *Replace* method, *InsertMasked* method

ResInstLoad

The *ResInstLoad* method loads a resource into the image list.

```
function ResInstLoad(Instance: THandle; ResType: TResType; Name: string; MaskColor: TColor): Boolean;
```

Description Use the *ResoInstLoad* method to load a resource of type *TResType* into the image list. The *MaskColor* is used to create the mask for the resulting image. *ResInstLoad* should be used rather than *ResouceLoad* to load a resource from within a package.

See also *GetInstRes* method, *GetResouce* method

ResourceLoad

The *ResourceLoad* method loads a resource into the image list.

```
function ResourceLoad(ResType: TResType; Name: string; MaskColor: TColor): Boolean;
```

Description Use the *ResourceLoad* method to load a resource of type *TResType* into the image list. The *MaskColor* is used to create the mask for the resulting image.

Note Within a package, *ResouceLoad* will return an error. *ResInstLoad* must be used to load a resource from within a package.

See also *GetResource* method, *ResInstLoad* method

SetDragImage

The *SetDragImage* method sets an image within the image list to be shown during a drag and drop operation.

```
function SetDragImage(Index, HotSpotX, HotSpotY: Integer): Boolean;
```

Description Use the *SetDragImage* method to set the image specified by *Index* within the image list to be shown when the image list is being used for drag and drop. The hot spot of the image is specified by *HotSpotX* and *HotSpotY*.

See also *DragMove* method, *BeginDrag* method, *DragLock* method, *DragUnlock* method, *EndDrag* method, *ShowDragImages* method

ShowDragImage

The *ShowDragImage* method shows the current drag image if it was previously hidden.

```
procedure ShowDragImage;
```

Description Use *ShowDragImage* method to show the current drag image if it was previously hidden.

See also *HideDragImage* method, *SetDragImage* method

UnRegisterChanges

The *UnRegisterChanges* method removes an object from the list of objects to be notified when the image list changes.

```
procedure UnRegisterChanges(Value: TChangeLink);
```

Description Use the *UnRegisterChanges* method to unregister a *TChangeLink* object. The object will no longer receive notification when the image list changes.

See also *RegisterChanges* method

TCustomLabel

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TGraphicControl • TCustomLabel

TCustomLabel is the base class for non-windowed controls that display text on a form.

Description Use *TCustomLabel* as a base class when defining objects that display text that cannot be edited by the user. *TCustomLabel* introduces several new properties and methods to control the appearance of the text. *TCustomLabel* is not a descendant of *TWinControl*, so it does not have its own window and can't receive direct input from the keyboard. To define an object that responds to direct keyboard input in addition to displaying text, use *TCustomStaticText* as a base class.

Do not create instances of *TCustomLabel*. To put a label on a form, use a *TLabel* object.

See also *TCustomStaticText* object, *TLabel* object, *TWinControl* object

C

Properties

The following table lists the properties of *TCustomLabel*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
Alignment	protected	taLeftJustify	TCustomLabel
AutoSize	protected	True	TCustomLabel
BoundsRect	public		TControl
Canvas	public, read-only		TGraphicControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TControl
ClientRect	public, read-only		TControl
ClientWidth	public		TControl

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
FocusControl	protected		TCustomLabel
Height	published		TControl
Hint	published		TControl
Layout	protected	tlTop	TCustomLabel
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ShowAccelChar	protected	True	TCustomLabel
ShowHint	public		TControl
Tag	published	0	TComponent
Top	published		TControl
Transparent	protected	False	TCustomLabel
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl
WordWrap	protected	False	TCustomLabel

Alignment

Alignment controls the horizontal placement of the text within the label.

```
type TAlignment = (taLeftJustify, taRightJustify, taCenter);
property Alignment: TAlignment;
```

Description Set *Alignment* to specify how the text of the label is aligned within the *ClientRect* of the label control. *Alignment* can be one of the following values:

Value	Meaning
taLeftJustify	The text appears lined up along the left edge of the label.

Value	Meaning
taRightJustify	The text appears lined up along the right edge of the label.
taCenter	The text is horizontally centered in the label.

The effect of the *Alignment* property is more obvious if the *WordWrap* property is *True* and the label includes more than one line of text.

See also *ClientRect* property, *GetLabelText* method, *Layout* property, *WordWrap* property

AutoSize

AutoSize determines whether the size of the label automatically resizes to accommodate the text.

```
property AutoSize: Boolean;
```

Description Use *AutoSize* to make the label adjust its size automatically so the client area accommodates the height and width of the text. When *AutoSize* is *False*, the label is fixed in size. When *AutoSize* is *True*, the size of the label is readjusted whenever its text changes. The size of the label is also readjusted when the *Font* property changes.

When *WordWrap* is *True*, the width of the label is fixed. If *AutoSize* is also *True*, changes to the text cause the label to change in height. When *AutoSize* is *True* and *WordWrap* is *False*, the height of the label is determined by the font, and changes to the text cause the label to change in width.

See also *BoundsRect* property, *GetLabelText* method, *TControl.Font* property, *WordWrap* property

C

FocusControl

FocusControl is a windowed control associated with the label.

```
property FocusControl: TWinControl;
```

Description Set *FocusControl* to the windowed control that should receive focus when the user presses the accelerator key specified by the label. Specify an accelerator key by preceding a character in the label text with an ampersand (&) and setting the *ShowAccelChar* property to *True*.

See also *ShowAccelChar* property, *GetLabelText* method, *TWinControl* object

Layout

Layout specifies the vertical placement of the text within the label.

```
type TTextLayout = (tlTop, tlCenter, tlBottom);
```

```
property Layout: TTextLayout;
```

Description Set *Layout* to specify how the text of the label is placed within the *ClientRect* of the label control. *Layout* can be one of the following values:

Value	Meaning
tlTop	The text appears at the top of the label.
tlCenter	The text is vertically centered in the label.
tlBottom	The text appears along the bottom of the label.

See also *Alignment* property, *ClientRect* property

ShowAccelChar

ShowAccelChar determines how an ampersand in the label text is displayed.

```
property ShowAccelChar: Boolean;
```

Description Set *ShowAccelChar* to *True* to allow the label to display an underlined accelerator key value. When *ShowAccelChar* is *True*, any character preceded by an ampersand (&) appears underlined. If the *FocusControl* property is set, the windowed control specified by the *FocusControl* property receives input focus when the user types that underlined character. To display an ampersand when *ShowAccelChar* is *True*, use two ampersands (&&) to stand for the single ampersand that is displayed.

Set *ShowAccelChar* to *False* to display the label text with all ampersands appearing as ampersands. When *ShowAccelChar* is *False*, the value of the *FocusControl* property is not used.

See also *FocusControl* property, *GetLabelText* method

Transparent

Transparent specifies whether controls that sit below the label on a form can be seen through the label.

```
property Transparent: Boolean;
```

Description Set *Transparent* to *True* to prevent the label from obscuring other controls on the form. For example, if the label is used to add text to a graphic, set *Transparent* to *True* so that the label does not stand out as a separate object.

Writing text so that the background is transparent is slower than writing text when *Transparent* is *False*. If the label is not obscuring a complicated image, performance can be improved by setting the background color of the label to match the object beneath it and setting *Transparent* to *False*.

See also *TControl.Color* property, *TControl.ParentColor* property

WordWrap

WordWrap specifies whether the label text wraps when it is too long for the width of the label.

```
property WordWrap: Boolean;
```

Description Set *WordWrap* to *True* to allow the label to display multiple line of text. When *WordWrap* is *True*, text that is too wide for the label control wraps at the right margin and continues in additional lines.

Set *WordWrap* to *False* to limit the label to a single line. When *WordWrap* is *False*, text that is too wide for the label appears truncated.

See also *ClientWidth* property, *GetLabelText* method

Methods

The following table lists the methods of *TCustomLabel*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
Create	public	TCustomLabel
DefaultHandler	public	TObject
Destroy	public	TGraphicControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent

Method	Visibility	Documented in
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetLabelText	protected	TCustomLabel
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTextBuf	public	TControl
GetTextLen	public	TControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
Invalidate	public	TControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TCustomLabel
Paint	protected	TCustomLabel
Perform	public	TControl
Refresh	public	TControl
RemoveComponent	public	TComponent
Repaint	public	TControl
SafeCallException	public	TComponent
ScreenToClient	public	TControl
SendToBack	public	TControl
SetAutoSize	protected	TCustomLabel
SetBounds	public	TControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TControl

Create

Create creates and initializes an instance of *TCustomLabel*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create an instance of *TCustomLabel* at runtime. Labels placed on forms at design time are created automatically.

After calling the inherited constructor, *Create* initializes the following properties:

- *AutoSize* to *True*.
- *ShowAccelChar* to *True*.
- *ControlStyle* to [*csOpaque, csReplicatable*].
- *Width* to 65 and *Height* to 17.

See also *AutoSize* property, *ControlStyle* property, *Height* property, *ShowAccelChar* property, *Width* property

GetLabelText

GetLabelText returns the value of the *Caption* property.

```
function GetLabelText: string; virtual;
```

Description Call *GetLabelText* to obtain the string that appears as the text of the label. *GetLabelText* is exposed as a protected method so that descendants can override this method to display text other than the value of the *Caption* property.

See also *TControl.Caption* property

Notification

Notification updates the *FocusControl* property when the windowed control it refers to is destroyed.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description *Notification* is called automatically when components are created or destroyed. After calling the inherited method, *Notification* checks whether the windowed control specified by the *FocusControl* property is about to be destroyed. If the windowed control is going away, *Notification* sets the *FocusControl* property to **nil**.

See also *FocusControl* property

Paint

Paint draws the image of the label.

```
procedure Paint; override;
```

Description *Paint* is called automatically when the label receives a WM_PAINT message from its *Parent*. *Paint* writes the text returned by the *GetLabelText* method.

See also *GetLabelText* method

SetAutoSize

SetAutoSize sets the *AutoSize* property.

```
procedure SetAutoSize(Value: Boolean); virtual;
```

Description *SetAutoSize* is the protected access method for the *AutoSize* property. It is exposed as a protected method so that descendants can override it to change the implementation of the *AutoSize* property.

See also *AutoSize* property

TCustomListBox

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomListBox

TCustomListBox is the abstract base class for objects that are wrappers for Windows list box controls.

Description *TCustomListBox* is an abstract class that encapsulates behavior common to all list box controls in Delphi. List box objects are used to display a scrollable list from which users can select one or more items. Use a list box control to limit the user to selecting from a list of items. Items can be added at design time or at runt time.

As a wrapper for a list box *TCustomListBox* introduces methods for:

- Displaying, sorting, selecting, manipulating and querying items in the list.
- Creating owner-draw style list boxes that can display graphical images such as icons and bitmaps.
- Creating multiple-column and multiple-selection lists.

By default *TCustomListBox* is a standard list box that displays items as text strings. These items are contained in a *TStrings* object, and the methods of the *TStrings* object are used to add, insert or delete a string. The following code demonstrates adding a new string to the list box:

```
MyListBox.Items.Add('New item');
```

List box descendents of *TCustomListBox* inherit the basic functionality of *TCustomListBox*. Some descendents also have specialized behavior and can be linked to file, directory, drive or database information. Use these list boxes to easily access and list this type of information.

A combo box is a list box with an edit control and an optional drop down style for the list. Use a combo box to allow users to type their selection into an edit box or to save space on a form by having the list drop down.

For lists of items that require a matrix or expandable tree-like structure use outlines, list views or tree views. These controls allow the grouping of items under column headings or in expandable and contractible rows.

Note For component writers wanting to create a customized list box objects, use *TCustomListBox* as a base class for deriving specialized list box objects.

See also *TStrings* object

Properties

The following table lists the properties of *TCustomListBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BorderStyle	protected	bsSingle	TCustomListBox
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomListBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Columns	protected	0	TCustomListBox
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
ExtendedSelect	protected	True	TCustomListBox
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
IntegralHeight	protected	False	TCustomListBox
ItemHeight	protected		TCustomListBox
ItemIndex	public		TCustomListBox
Items	public		TCustomListBox
Left	published		TControl
MultiSelect	protected	False	TCustomListBox
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	False	TControl

C

Property	Visibility	Default	Documented in
ParentWindow	public		TWinControl
SelCount	public, read-only		TCustomListBox
Selected	public		TCustomListBox
ShowHint	public		TControl
Showing	public, read-only		TWinControl
Sorted	protected	False	TCustomListBox
Style	protected	lbStandard	TCustomListBox
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
TabWidth	protected	0	TCustomListBox
Tag	published	0	TComponent
Top	published		TControl
TopIndex	public		TCustomListBox
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

BorderStyle

BorderStyle determines whether the list box has a border.

```
type TBorderStyle = bsNone..bsSingle;
```

```
property BorderStyle: TBorderStyle;
```

Description Use *BorderStyle* to change the appearance of the outer boundary of the list box control. A list box can have a single border or no visible border. The following table lists the possible values for *TBorderStyle,* which is the type of the *BorderStyle* property:

Value	Meaning
bsNone	No visible border
bsSingle	Single-line border

See also *Ctl3D* property

Canvas

Canvas provides access to a drawing surface to use when implementing a handler for the *OnDrawItem* event of an owner-draw list box.

```
property Canvas: TCanvas;
```

Description *Canvas* is only useful for applications that control the drawing of list box items, that is, for owner-draw list boxes. *Canvas* is used to draw graphical images that can be used as items in a list box.

By default a *TControlCanvas* object is created for the *Canvas* property in the list box constructor. To create an owner-draw list box that uses a canvas, set the *Style* property accordingly.

Example The following code draws a graphic stored in the *Objects* property of the *Items* list of *ListBox1*. This code would be attached to the *OnDrawItem* event handler of *ListBox1*.

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
   Rect: TRect; State: TOwnerDrawState);
var
   SourceRect: TRect;
begin
   SourceRect.Top := 0;
   SourceRect.Left := 0;
   SourceRect.Bottom := TBitmap(ListBox1.Items.Objects[Index]).Height;
   SourceRect.Right := TBitmap(ListBox1.Items.Objects[Index]).Width;
   ListBox1.Canvas.CopyRect(Rect, TBitmap(ListBox1.Items.Objects[Index]).Canvas,
SourceRect);
end;
```

See also *ItemHeight* property, *OnDrawItem* event, *Style* property, *TCanvas* object, *TControlCanvas* object

Columns

Columns specifies the number of columns, in a multi-column list box, that are visible without having to scroll.

```
property Columns: Integer;
```

Description Use *Columns* to specify the number of columns, in a multi-column list box, that are visible without having to use the horizontal scrollbar.

Multi-column list boxes have a horizontal scrollbar that allows users to view multiple columns as they wrap. The default value for *Columns* is 0, meaning that the list box is not multi-column. That is, users can scroll only vertically and the list of items will not wrap.

For *Columns* values greater than 0, multiple columns accommodate the items as they wrap beyond the bottom of the list box. The *Columns* property specifies the number of columns that are visible without having to horizontally scroll the list box.

The width of each column depends upon both the *Width* property and the number of *Columns*.

See also *ItemHeight* property, *Width* property

ExtendedSelect

ExtendedSelect determines if the user can select a sequential range of items in the list box.

property ExtendedSelect: Boolean;

Description *ExtendedSelect* works in conjunction with the *MultiSelect* property. If *ExtendedSelect* is *True* and *MultiSelect* is *True*, the user can select an item then hold down the *Shift* key and select another and all the items in between the two selected items also become selected. If the user doesn't hold down the *Shift* or *Ctrl* key while selecting a second item, the first selected item becomes unselected—in other words, the user must use the *Ctrl* key to select multiple noncontiguous items, or the *Shift* key to select a range of items.

If *ExtendedSelect* is *False* and *MultiSelect* is *True*, the user can select multiple items without using the *Shift* or *Ctrl* key, but they can't select a range of items in one operation.

If *MultiSelect* is *False*, the setting of *ExtendedSelect* has no effect as the user will not be able to select more than one item at a time in the list box.

See also *MultiSelect* property

IntegralHeight

IntegralHeight determines whether the list box displays the partial items.

property IntegralHeight: Boolean;

Description *IntegralHeight* controls the height of the list box itself.

When *IntegralHeight* is *True*, and the Style property is *lsOwnerDrawFixed*, the list box height will always be a multiple of the *ItemHeight* property value. In this case items will never be partially displayed.

If *IntegralHeight* is *False*, the list box height can be set to any value, but the last item may not be fully displayed.

If the *Style* property value of the list box is *lbOwnerDrawVariable*, setting the *IntegralHeight* property to *True* has no effect.

See also *ItemHeight* property, *Items* property, *Style* property

ItemHeight

ItemHeight specifies the height in pixels of an item in an owner-draw list box.

property ItemHeight: Integer;

Description Use *ItemHeight* to control the height of an item in a list box when the *Style* property is *lsOwnerDrawFixed*. The *ItemHeight* property can also affect the height of the list box itself. When the *IntegralHeight* property is *True*, the height of the list box is always an increment of the *ItemHeight* value.

If the *Style* property is *lsStandard* or *lsOwnerDrawVariable*, the value of *ItemHeight* is ignored.

See also *DrawItem* method, *IntegralHeight* property, *Items* property, *MeasureItem* method, *OnDrawItem* event, *OnMeasureItem* event, *Style* property

ItemIndex

ItemIndex specifies the ordinal number of the selected item in the list box's item list.

property ItemIndex: Integer;

Description Use *ItemIndex* to select an item at runtime. Set the value of *ItemIndex* to the index of the item to be selected. The *ItemIndex* of the first item in the list box is 0. If no item is selected, the value is –1, which is the default value unless *MultiSelect* is *True*.

If the value of the *MultiSelect* property is *True* the user can select more than one item in the list box. In this case, the *ItemIndex* value is the index of the selected item that has focus. If *MultiSelect* is *True*, *ItemIndex* defaults to 0.

See also *Items* property, *MultiSelect* property

Items

Items contains the strings that appear in the list box.

property Items: TStrings;

Description Use *Items* to add, insert, delete and move items. By default the items in a list box are of type *TStrings*. Use this item type to access its methods or properties in order to manipulate the items in the list. For example, the following code adds the text in the edit box to the list box as an item:

```
ListBox1.Items.Add(Edit1.Text);
```

See also *Clear* method, *ItemIndex* property, *OnDrawItem* event, *Selected* property, *TStrings* object

MultiSelect

MultiSelect determines whether the user can select more than one element at a time from the list box.

property MultiSelect: Boolean;

Description Set *MultiSelect* to *True* to allow the user to select multiple items. If *MultiSelect* if *False*, multiple items cannot be selected in the list box at the same time.

MultiSelect allows users to select multiple non-sequential items. It does not allow users to select a range of items in one operation, as does the *ExtendedSelect* property.

When *MultiSelect* is *True* and multiple items are selected, the value of the *ItemIndex* property, which indicates the index of the *Selected* item, is the index of the selected item that has focus.

See also *ExtendedSelect* property, *ItemIndex* property, *SelCount* property, *Selected* property

SelCount

SelCount reports the number of items that are selected in a list box when multiple selections are allowed.

```
property SelCount: Integer;
```

Description *SelCount* is read-only. Use *SelCount* to find the number of selected items in the list box when the *MultiSelect* property is *True*.

When the *MultiSelect* property is *False*, *SelCount* is always –1.

See also *ExtendedSelect* property, *MultiSelect* property, *Selected* property

Selected

Selected determines whether a particular item is selected in a list box.

```
property Selected[Index: Integer]: Boolean;
```

Description Use *Selected* to query the state of a particular item in the list box. If the item specified by the *Index* parameter is selected in the list box, the value of the *Selected* property is *True*. If the specified item is not selected, *Selected* is *False*.

The *Index* parameter is the item referenced by its position in the list box, with the first item having an *Index* value of 0.

An item is selected when the user highlights it. More than one item in the list box can be selected by setting the *MultiSelect* property to *True*. In this case, the *Selected* item is the one that has focus.

See also *ExtendedSelect* property, *Items* property, *MultiSelect* property, *SelCount* property

Sorted

Sorted specifies whether the items in a list box are arranged alphabetically.

```
property Sorted: Boolean;
```

Description Use *Sorted* to sort the items by setting its value to *True*. If *Sorted* is *False*, the items are unsorted. When *Sorted* is *True*, items that are added or inserted are automatically into the list box in alphabetical order.

See also *Items* property

Style

Style determines whether the list box is standard or owner-draw.

```
type TListBoxStyle = (lbStandard, lbOwnerDrawFixed, lbOwnerDrawVariable);

property Style: TListBoxStyle;
```

Description Use *Style* to specify whether the list box is a standard list box that displays text strings, or an owner-draw list box that can display graphical images. By default, *Style* is *lbStandard*, meaning that the list box displays each item as a string.

Owner-draw list boxes gives the user a chance to display items in some non-standard way. In this case the user must write the code. To create owner-draw list boxes, set the value of *Style* to *lbOwnerDrawFixed* or *lbOwnerDrawVariable*. The following table lists the values for the *TListBoxStyle* type of the Style property:

Value	Meaning
lbStandard	All items are strings, with each item the same height.
lbOwnerDrawFixed	Each item in the list box is the height specified by the *ItemHeight* property.
lbOwnerDrawVariable	Items in the list box can be of varying heights.

Owner-draw list boxes require more programming because the application needs information on how to render the image for each item in the list.

Each time an item is displayed in an *lbOwnerDrawFixed* list boxes, the *OnDrawItem* event occurs. The event handler for *OnDrawItem* draws the specified item. The *ItemHeight* property determines the height of each of the items.

Each time an item is displayed in an *lbOwnerDrawVariable* list box, two events occur. The first is the *OnMeasureItem* event. The code for the *OnMeasureItem* handler can set the height of each item. Then the *OnDrawItem* event occurs. The code for the *OnDrawItem* handler draws each item in the list box using the size specified by the *OnMeasureItem* handler.

See also *DrawItem* method, *ItemHeight* property, *Items* property, *MeasureItem* method, *OnDrawItem* event, *OnMeasureItem* event

TabWidth

TabWidth specifies the size of the tabs in the list box.

```
property TabWidth: Integer;
```

Description Set *TabWidth* to control the number of dialog base units displayed for each tab character that appears in the strings of a list box.

If *TabWidth* is set to 0, which is the default setting, the tabs default to the width specified by Windows, which is two dialog base units.

TopIndex

TopIndex specifies the index number of the item that appears at the top of the list box.

```
property TopIndex: Integer;
```

Description Use *TopIndex* property to find or set the first item displayed at the top of the list box. *TopIndex* can be used, for example, to change the topmost item to a different item in the list.

Example The following example uses a list box containing a list of strings, a button, and an edit box on a form. When the user runs the application and clicks the button, the third item in the list becomes the first item, and the index value of that item appears in the edit box. The index value displayed is 2, indicating the third item in the list (the first item in the list has an index value of 0):

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Number: Integer;
begin
  for Number := 1 to 20 do
    ListBox1.Items.Add('Item ' + IntToStr(Number));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.TopIndex := 2;
  Edit1.Text := IntToStr(ListBox1.TopIndex);
end;
```

See also *ItemIndex* property, *Items* property, *Sorted* property

Events

The following table lists the events of *TCustomListBox*:

Event	Visibility	Documented in
OnDrawItem	protected	TCustomListBox
OnMeasureItem	protected	TCustomListBox

OnDrawItem

OnDrawItem occurs whenever an item in an owner-draw list box needs to be redisplayed.

```
type
  TOwnerDrawState = set of (odSelected, odGrayed, odDisabled, odChecked, odFocused);
  TDrawItemEvent = procedure(Control: TWinControl; Index: Integer; Rect: TRect; State:
    TOwnerDrawState) of object;

property OnDrawItem: TDrawItemEvent;
```

Description Use *OnDrawItem* to write a handler for drawing of the items in list boxes with the *Style* values *lbOwnerDrawFixed* or *lbOwnerDrawVariable*. *OnDrawItem* occurs whenever the list box needs to display an item. *OnDrawItem* occurs only for owner-draw list boxes.

OnDrawItem is of type *TDrawItemEvent* and passes four parameters to its handler describing the item to be drawn:

- *Control*, specifying a reference to the control containing the item.
- *Index*, specifying the index of the item in that control.
- *Rect*, specifying a rectangle in which to draw.
- *State*, specifying the state of the item (selected, focused or disabled).

The size of the rectangle that contains the item is determined either by the *ItemHeight* property for fixed owner-draw list boxes or by the response to the *OnMeasureItem* event for variable owner-draw list boxes.

The *Index* parameter is the position of the item in the list box.

Rect is the area in the list box where the item is to be drawn. The *Rect* parameter automatically provides the proper location of the item within the control's canvas.

State is the current state of the item in the list box. The *TOwnerDrawState* type defines the possible values for the *State* parameter. The following table lists these values:

Value	Meaning
odSelected	The item is selected.
odDisabled	The entire list box is disabled.
odFocused	The item currently has focus.

Example Here is a typical handler for an *OnDrawItem* event. In the example, a list box with the *lbOwnerDrawFixed* style draws a bitmap to the left of each string.

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
   Rect:TRect;State: TOwnerDrawState);
var
   Bitmap: TBitmap;          { temporary variable for the item's bitmap }
   Offset: Integer;          { text offset width }
begin
   with (Control as TListBox).Canvas do{ draw on control canvas, not on the form }
   begin
   FillRect(Rect);           { clear the rectangle }
   Offset := 2;              { provide default offset }
   Bitmap := TBitmap((Control as TListBox).Items.Objects[Index]);{ get the bitmap }
   if Bitmap <> nil then
   begin
      BrushCopy(Bounds(Rect.Left + 2, Rect.Top, Bitmap.Width, Bitmap.Height),
         Bitmap, Bounds(0, 0, Bitmap.Width, Bitmap.Height), clRed);{render bitmap}
      Offset := Bitmap.width + 6;{ add four pixels between bitmap and text}
   end;
   TextOut(Rect.Left + Offset, Rect.Top, (Control as TListBox).Items[Index]){
display the text }
```

```
    end;
  end;
```

See also *DrawItem* method, *ItemHeight* property, *MeasureItem* method, *OnMeasureItem* event, *Style* property

OnMeasureItem

OnMeasureItem occurs whenever an application needs to redisplay an item in a variable height owner-draw list box.

```
type TMeasureItemEvent = procedure(Control: TWinControl; Index: Integer; var Height: Integer)
  of object;

property OnMeasureItem: TMeasureItemEvent;
```

Description Use *OnMeasureItem* to write a handler to measure items to be drawn in a list box with a *Style* property value of *lbOwnerDrawVariable*.

OnMeasureItem is of type *TMeasureItemEvent* which contains three parameters describing the item to measure:

- *Control*, specifying the control containing the item.
- *Index*, specifying the index of the item in the control.
- *Height*, specifying the height of the item.

The *Index* parameter identifies the position of the item in the list box.

The *Height* parameter should specify the height in pixels that the given item will occupy in the control. The *Height* parameter is a *var* parameter, which initially contains the default height of the item or the height of the item text in the control's font. The handler can set *Height* to a value appropriate to the contents of the item, such as the height of a graphical image to be displayed within the item.

After the *OnMeasureItem* event occurs, the *OnDrawItem* event occurs, rendering the item with the measured size.

Example Here is a typical handler for an *OnMeasureItem* event. The example assumes that a variable owner-draw list box already has bitmaps associated with each of its strings. It sets the height of the item to the height of the associated bitmap if that height is greater than the default height.

```
procedure TForm1.ListBox1MeasureItem(Control: TWinControl; Index: Integer;
  var Height: Integer);
var
  Bitmap: TBitmap;
begin
  with Control as TListBox do
  begin
    Bitmap := TBitmap(Items.Objects[Index]);
    if Bitmap <> nil then
    if Bitmap.Height > Height then Height := Bitmap.Height;
  end;
end;
```

See also *DrawItem* method, *ItemHeight* property, *MeasureItem* method, *OnDrawItem* event, *Style* property

Methods

The following table lists the methods of *TCustomListBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomListBox
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomListBox
CreateParams	protected	TCustomListBox
CreateParented	public	TWinControl
CreateWnd	protected	TCustomListBox
DefaultHandler	public	TObject
DeleteString	protected	TCustomListBox
Destroy	public	TCustomListBox
DestroyComponents	public	TComponent
Destroying	public	TComponent
DestroyWnd	protected	TCustomListBox
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragCanceled	protected	TCustomListBox
DragDrop	public	TControl
Dragging	public	TControl
DrawItem	protected	TCustomListBox
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent

Method	Visibility	Documented in
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetItemData	protected	TCustomListBox
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
ItemAtPos	public	TCustomListBox
ItemRect	public	TCustomListBox
MeasureItem	protected	TCustomListBox
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
ResetContent	protected	TCustomListBox
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl

Method	Visibility	Documented in
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetItemData	protected	TCustomListBox
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl
WndProc	protected	TCustomListBox

Clear

Clear deletes all items from the list box.

procedure Clear;

Description Use *Clear* to delete all of the items in the list box at once.

Although the items in a standard list box are, by default, of type *TStrings*, *Clear* allows the user to delete all of the items without referring to the item type. For example, the code to use the *Clear* method of the list box is simply:

```
ListBox1.Clear;
```

rather than

```
ListBox1.Items.Clear;
```

which calls the item object's *Clear* method and also clears the items.

See also *TStrings.Clear* method, *Items* property

C

Create

Create instantiates a list box control.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a list box control at runtime. For list boxes created at design time, *Create* is called automatically.

Create allocates memory for a list box control, and calls the inherited *Create*. Then it:

- Initializes values for its *Width*, *Height* and *ItemHeight* properties.
- Sets *ParentColor* to *False*, and *TabStop* and *ExtendedSelect* to *True*.
- Sets the *ControlStyle* for the list box.
- Sets the *BorderStyle* property to *bsSingle*.
- Creates a *TControlCanvas* object for the *Canvas* property.
- Creates a *TStrings* type of object for the *Items* property.

See also *BorderStyle* property, *Canvas* property, *ControlStyle* property, *ExtendedSelect* property, *Height* property, *ItemHeight* property, *Items* property, *ParentColor* property, *TabStop* property, *TStrings* object, *Width* property

CreateParams

CreateParams initializes the window-creation parameter record passed in the *Params* parameter.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;
procedure CreateParams(var Params: TCreateParams); virtual;
```

Description *CreateParams* is used to pass information, in its *Params* parameter, to *CreateWindowHandle* when the list box control is created. The list box *CreateWnd* method calls the inherited *CreateWnd* which passes this information in its call to *CreateWindowHandle* (a protected method inherited from *TWinControl*).

TCustomListBox overrides *CreateParams* to customize the representation of the window. It uses information from the list box *Columns*, *ExtendedSelect*, *IntegralHeight*, *MultiSelect*, *Sorted* and *Style* properties and sets WS_HSCROLL, WS_VSCROLL, LBS_HASSTRINGS and LBS_NOTIFY Windows styles for creation of the list box.

When overriding *CreateParams*, always call the inherited method first to set the default values, then make any desired adjustments.

TCreateParams is the type of the *Params* parameter for *CreateParams*. It is a data structure holding information pertinent to creating a window handle. *CreateWindowHandle* processed he fields of the *TCreateParams* record to become the parameters for a call to the *CreateWindowEx* API function.

See also *Columns* property, *TWinControl.CreateWindowHandle* method, *TWinControl.CreateWnd* method, *ExtendedSelect* property, *IntegralHeight* property, *MultiSelect* property, *Sorted* property, *Style* property

CreateWnd

CreateWnd creates a Windows control corresponding to the list box component.

procedure CreateWnd; **virtual**;

Description *CreateWnd* is called automatically by *CreateHandle* (a protected method inherited from *TWinControl*). *CreateWnd* calls the inherited *CreateWnd*, which calls:

- *CreateParams* to initialize the window-creation parameters.
- *CreateWindowHandle* to create the window handle for the control.

TCustomListBox CreateWnd additionally sets up the list box position, tabs, items, and columns.

See also *Columns* property, *TWinControl.CreateHandle* method, *CreateParams* method, *TWinControl.CreateWindowHandle* method, *TWinControl.CreateWnd* method, *ItemIndex* property, *Items* property, *TabStops* property, *TabWidth* property, *TopIndex* property

DeleteString

DeleteString deletes a string from the list box.

procedure DeleteString(Index: Integer); **dynamic**;

Description *DeleteString* deletes the string, specified by the *Index* parameter, from list box control.

Destroy

Destroy disposes of the instance of a list box control.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, an application should call *Free*. Free verifies that the list box object has not already been freed before it calls *Destroy*.

Destroy frees the memory associated with the objects for the *Canvas* and *Items* properties. Then it calls the inherited *Destroy* to destroy the list box object.

See also *Free* method

DestroyWnd

DestroyWnd saves the list box *Items* in a string list before calling the inherited *DestroyWnd*.

procedure DestroyWnd; **virtual**;

Description *DestroyWnd* saves the list box text string items, in the *Items* property, to internal storage in a *TStringList* object that it creates. Then it calls the inherited *DestroyWnd* to destroy the windowed control's window handle and free associated device contexts.

See also *TWinControl.DestroyWnd* method, *Items* property, *TStringList* object

DragCanceled

DragCanceled sends a mouse message cancelling the drag-and-drop operation.

```
procedure DragCanceled; dynamic;
```

Description *DragCanceled* is called in response to the cancellation of a drag-and-drop operation that had started. Users can cancel the operation by dropping the item over a component that does not accept the item, or by pressing Esc while dragging.

Delphi calls the *DragCanceled* method before invoking the *OnEndDrag* event.

See also *TControl.OnEndDrag* event

DrawItem

DrawItem is the implementation method for the *OnDrawItem* event of owner-draw list boxes.

```
TPoint = record
  X: Longint;
  Y: Longint;
end;

TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;

TOwnerDrawState = set of (odSelected, odGrayed, odDisabled, odChecked, odFocused);

procedure DrawItem(Index: Integer; Rect: TRect; State: TOwnerDrawState); virtual;
```

Description *DrawItem* is called whenever a visual aspect of an owner-draw list box changes. The list box calls *DrawItem* for each visible item in its list, passing the index of the item in the *Index* parameter.

By default, the *DrawItem* method for a list box calls any event handler attached to the component's *OnDrawItem* event. If there is no handler attached, *DrawItem* fills the rectangle passed in *Rect* and draws any text associated with the indexed item. The default drawing ignores the *State* parameter, although *State* is passed along to attached event handlers.

See also *ItemHeight* property, *MeasureItem* method, *OnDrawItem* event, *OnMeasureItem* event, *Style* property

GetItemData

GetItemData returns a 32-bit value associated with the specified item in a list box.

function GetItemData(Index: Integer): LongInt; **dynamic**;

Description *GetItemData* is used to retrieve the data associated with a particular item in a list box. This value can be displayed as that list box item.

See also *SetItemData* method

ItemAtPos

ItemAtPos returns the index of the list box item indicated by the coordinates of a point on the control.

function ItemAtPos(Pos: TPoint; Existing: Boolean): Integer;

Description Use *ItemAtPos* to detect if an item exists at a particular point in the control.

The *Pos* parameter is the point in the control in window coordinates. If *Pos* is beyond the last item in the list box, the value of the *Existing* variable determines the returned value. If *Existing* is set to *True*, *ItemAtPos* returns –1, indicating that no item exists at that point. If *Existing* is set to *False*, *ItemAtPos* returns the index of the last item in the list box plus one.

See also *ItemIndex* property, *Items* property

ItemRect

ItemRect returns the rectangle that surrounds the item specified in the *Item* parameter.

```
TPoint = record
  X: Longint;
  Y: Longint;
end;
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
function ItemRect(Item: Integer): TRect;
```

Description Use *ItemRect* to get the coordinates of a particular item in the list box. For example, *ItemRect* is used internally by the *ItemAtPos* method.

The *Item* parameter is the *ItemIndex* of the item whose position is queried.

See also *ItemAtPos* method, *ItemIndex* property, *TRect* type

MeasureItem

MeasureItem is the implementation method for the *OnMeasureItem* event for an owner-draw list box.

procedure MeasureItem(Index: Integer; **var** Height: Integer); **virtual**;

Description *MeasureItem* is called whenever a visual aspect of a variable owner-draw list box, with the *csOwnerDrawVariable Style* set, changes.

Before drawing its items, a variable owner-draw list box calls *MeasureItem* once for each visible item, passing the index of the item to measure, and the default height of the item. Since *Height* is a *var* parameter, *MeasureItem* can increase or reduce the height of each item as needed.

By default the *MeasureItem* method does nothing except call any event handler attached to the *OnMeasureItem* event. Override *MeasureItem* to change or add functionality to the default behavior.

See also *DrawItem* method, *ItemHeight* property, *ItemIndex* property, *OnDrawItem* event, *OnMeasureItem* event, *Style* property

ResetContent

ResetContent removes all of the items from a list box.

```
procedure ResetContent; dynamic;
```

Description *ResetContent* is used to remove the items from the list box. *ResetContent* is used, for example, by the *Clear* method.

See also *Clear* method

SetItemData

SetItemData sets the 32-bit value associated with the specified item in a list box.

```
procedure SetItemData(Index: Integer; AData: LongInt); dynamic;
```

Description *SetItemData* is used to set the data associated with a particular item in a list box. This value can be displayed as that list box item.

See also *GetItemData* method

WndProc

WndProc provides some additional behavior for auto drag before calling the inherited method.

```
TMessage = record
  Msg: Word;
  case Integer of
    0: (
      WParam: Word;
      LParam: Longint;
      Result: Longint);
    1: (
      WParamLo: Byte;
      WParamHi: Byte;
```

```
    LParamLo: Word;
    LParamHi: Word;
    ResultLo: Word;
    ResultHi: Word);
end;

procedure WndProc(var Message: TMessage); virtual;
```

Description *WndProc* overrides the inherited *WndProc* to provide specialized responses to messages that allows the list box to handle auto drag mode itself. Then it calls the inherited *WndProc* to handle all other messages.

When overriding the *WndProc* method, be sure to call the inherited *WndProc* at the end to dispatch any other messages.

See also *TWinControl.WndProc* method

TCustomListView

Unit: ComCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomListView

The *TCustomListView* component is the abstract base type for all list view components, including *TListView*.

Description Use *TCustomListView* as a base class when defining a control that displays a list of items. *TCustomListView* allows a list of items to be displayed in columns with column headers and sub-items, or vertically or horizontally, with small or large icons.

Do not create instances of *TCustomListView*. To put a list view on a form, use a descendant of *TCustomListView*, such as *TListView*.

See also *TListColumn* object, *TListItem* object, *TListView* object

C

Properties

The following table lists the properties of *TCustomListView*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
AllocBy	protected	0	TCustomListView
BorderStyle	protected	bsSingle	TCustomListView
BoundingRect	public, read-only		TCustomListView
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Checkboxes	public	False	TCustomListView
ClientHeight	public		TControl

Property	Visibility	Default	Documented in
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Column	public, read-only		TCustomListView
ColumnClick	protected	True	TCustomListView
Columns	protected		TCustomListView
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
DropTarget	public		TCustomListView
Enabled	public	True	TControl
GridLines	public	False	TCustomListView
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
HideSelection	protected	True	TCustomListView
Hint	published		TControl
HotTrack	public	False	TCustomListView
IconOptions	protected		TCustomListView
ItemFocused	public		TCustomListView
Items	protected		TCustomListView
LargeImages	protected		TCustomListView
Left	published		TControl
MultiSelect	protected	False	TCustomListView
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ReadOnly	protected		TCustomListView
RowSelect	public	False	TCustomListView
SelCount	public, read-only		TCustomListView
Selected	public		TCustomListView
ShowColumnHeaders	protected	True	TCustomListView

Property	Visibility	Default	Documented in
ShowHint	public		TControl
Showing	public, read-only		TWinControl
SmallImages	protected		TCustomListView
SortType	protected	stNone	TCustomListView
StateImages	protected		TCustomListView
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
TopItem	public, read-only		TCustomListView
VCLComObject	public		TComponent
ViewOrigin	public, read-only		TCustomListView
ViewStyle	protected	vsIcon	TCustomListView
Visible	public	True	TControl
VisibleRowCount	public, read-only		TCustomListView
Width	published		TControl
WindowProc	public		TControl

AllocBy

AllocBy indicates the number of items the list view can store in memory.

property AllocBy: Integer;

Description Set *AllocBy* to the total number of items in the list before adding a large number of items to the list view. *AllocBy* allows the list view to allocate the memory for items in a single block at once rather than bit by bit as items are added.

See also *Items* property

BorderStyle

BorderStyle determines whether the list view displays a single line border.

property BorderStyle: TBorderStyle;

Description Use *BorderStyle* to affect the sharpness with which the list view control stands out on a form. *BorderStyle* can have a value of either *bsSingle* or *bsNone*. If *BorderSyle* is *bsSingle*, the list view is drawn with a single-line border. If *BorderStyle* is *bsNone*, there will be no border.

Note *BorderStyle* interacts with the *Ctl3D* property to create different visual effects. Try combining different combinations of the two properties to get the desired look.

See also *TWinControl.Ctl3D* property

BoundingRect

BoundingRect is the bounding rectangle for the items in the list view.

property BoundingRect: TRect;

Description Read *BoundingRect* to obtain the screen coordinates of a rectangle that encloses all the items in the list view. *BoundingRect* allows an application to adjust the size of the control to fit the items or to determine whether the mouse is over any of the list items.

Note To use *BoundingRect*, the list view's *ViewStyle* property must be either *vsIcon* or *vsSmallIcon*.

See also *ViewStyle* property

CheckBoxes

CheckBoxes specifies whether checkboxes appear next to the items in the list view.

property Checkboxes: Boolean;

Description Set *CheckBoxes* to *True* to make checkboxes appear next to the list items when *ViewStyle* is *vsList* or *vsReport*. Each line in the list displays a single check box.

See also *TListItem.Checked* property, *ViewStyle* property

Column

Column describes the column specified by the *Index* parameter.

property Column[**Index**: Integer]: TListColumn;

Description Read *Column* to access the *TListColumn* object that describes the column attributes of a column in a report style list view. The *Index* parameter is the index of the column in the *Columns* property array.

Note *Column* is only relevant when the value of the *ViewStyle* property is *vsReport*.

See also *Columns* property, *TListColumn* property, *ViewStyle* property

ColumnClick

ColumnClick determines whether the column header behaves like a button.

property ColumnClick: Boolean;

Description Use *ColumnClick* to specify whether the *OnColumnClick* event should occur when the user clicks on the column header. When *ColumnClick* is *True*, the *OnColumnClick* occurs when the column header is clicked with the mouse. When *ColumnClick* is *False*, the *OnColumnClick* event never occurs.

Note For column clicks to work, the column headers must be displayed. To display the headers, set the *ShowColumnHeaders* property to *True* and the *ViewStyle* property to *vsReport*.

See also *OnColumnClick* event, *ShowColumnHeaders* property, *ViewStyle* property

Columns

Columns describes the properties of the columns in the list view.

```
property Columns: TListColumns;
```

Description Use *Columns* to add or delete columns in the list view, or to edit their display properties. When setting *Columns* at design time a ListView Columns Editor dialog appears. This dialog allows columns to be added or deleted, and the display properties of individual columns to be changed. At runtime use the *Caption*, *Alignment*, and *Width* properties of the *TListColumn* objects accessed by the *Columns* property to change the appearance of the column headers.

Note To display the columns, set the *ViewStyle* property to *vsReport* and the *ShowColumnHeaders* property to *True*.

See also *ShowColumnHeaders* property, *TListColumns* object, *TListColumn* object, *ViewStyle* property

DropTarget

DropTarget specifies which item in the list is the target of a drag and drop operation.

```
property DropTarget: TListItem;
```

Description Read *DropTarget* to determine whether an item in the list is the target of a drag and drop operation. Set *DropTarget* to change whether a particular item in the list can be the drop target of a dragged item.

Note *DropTarget* only indicates the list item that appears to the user as a drop target. The application must still implement the drag-and-drop behavior for the item.

See also *TControl.EndDrag* method, *TListItem* object, *TListItem.DropTarget* property

GridLines

GridLines determines whether lines are drawn separating items in the list.

```
property GridLines: Boolean;
```

Description Set *GridLines* to *True* to add lines that separate the items in the list view. The value of *GridLines* is ignored unless *ViewStyle* is *vsList* or *vsReport*.

See also *ViewStyle* property

HideSelection

HideSelection determines whether the list view gives a visual indication of which item is selected when focus shifts to another control.

```
property HideSelection: Boolean;
```

Description Set *HideSelection* to *False* to provide visual feedback of the selected item in the list even when the list view does not have focus. Set *HideSelection* to *True* to show the selection only when the list view has focus. *HideSelection* does not affect the actual value of the selection, only the visual indication. Always setting *HideSelection* to *False* can make forms with many list views look too busy.

See also *ItemFocused* property, *MultiSelect* property, *SelCount* property, *Selected* property

HotTrack

HotTrack specifies whether list items are highlighted when the mouse passes over them.

```
property HotTrack: Boolean;
```

Description Set *HotTrack* to *True* to have the list view provide visual feedback about which item is under the mouse. When *HotTrack* is *False*, the list view items do not change color when the mouse passes over them.

See also *HideSelection* property

IconOptions

IconOptions determines how icons displayed in the list view are arranged.

```
property IconOptions: TIconOptions;
```

Description Set the properties of the *TIconOptions* object that is the value of *IconOptions* to specify how the icons in the list view should be arranged. The *ViewStyle* of the list view must be set to either *vsIcon* or *vsSmallIcons* for these options to affect the control.

See also *TIconOptions* object, *ViewStyle* property

ItemFocused

ItemFocused indicates which item, if any, has focus.

```
property ItemFocused: TListItem;
```

Description Read *ItemFocused* to determine which item, if any, can be edited by the user. Set *ItemFocused* to set focus to an item in the list. When an item has focus, it appears surrounded by a standard focus rectangle. If no item has focus, *ItemFocused* returns **nil**. Only one item can have focus at a time.

See also *Selected* property

Items

Items contains the list of items displayed by the list view.

property Items: TListItems;

Description Use *Items* to directly access the *TListItem* objects that represent the items in the list. Setting this property at design time brings up the ListView Items Editor. Use this dialog to add or delete items or subitems, and to edit their display properties. At runtime, use each item's *Caption*, *ImageIndex* and *StateIndex* properties to change the appearance of the list items.

See also *TListItem* object, *TListItems* object

LargeImages

LargeImages provides a list of icon images to display for each item in the list when *ViewStyle* is *vsIcon*.

property LargeImages: TImageList;

Description Set *LargeImages* to specify the bitmaps that should be displayed next to items in the list when *ViewStyle* is *vsIcon*. Each item in the *Items* list can be associated with a bitmap in this image list by setting its *ImageIndex* property.

See also *Items* property, *SmallImages* property, *StateImages* property, *TImageList* object, *ViewStyle* property

MultiSelect

MultiSelect determines whether the user can select more than one list item at a time.

property MultiSelect: Boolean;

Description Set *MultiSelect* to specify whether users can multiselect items in the list. If *MultiSelect* is *True*, the user can select multiple items. If *MultiSelect* if *False*, only one item can be selected at a time.

See also *SelCount* property, *Selected* property

ReadOnly

ReadOnly determines whether the user can change the contents of the control.

property ReadOnly: Boolean;

Description Use *ReadOnly* to determine whether the list view can be used to enter data. If *ReadOnly* is *True*, the list view displays values that the user can't edit. If *ReadOnly* is *False*, the user can modify the contents. If *ReadOnly* is *False*, the

OnEditing and *OnEdited* events occur when the user begins and finishes editing the list item, respectively.

See also *OnEdited* event, *OnEditing* event

RowSelect

RowSelect determines whether an entire row in the list view can be selected.

```
property RowSelect: Boolean;
```

Description Set *RowSelect* to *True* to allow entire rows to be selected when the *ViewStyle* property is *vsReport*. To allow multiple rows to be selected, use the *MultiSelect* property.

See also *MultiSelect* property, *TListItem.Selected* property, *ViewStyle* property

SelCount

SelCount is the number of selected items in the list view.

```
property SelCount: Integer;
```

Description Read *SelCount* to determine the number of items that are currently selected. When *SelCount* is 1, use the *Selected* property to access the selected item. To locate the selected items when *SelCount* is greater than one, check the *Selected* property of the list items, beginning with the item specified by the *Selected* property.

Note *SelCount* can only exceed 1 if the *MultiSelect* property is *True*.

See also *Items* property, *MultiSelect* property, *Selected* property, *TListItem.Selected* property

Selected

Selected is the first selected item in the list view.

```
property Selected: TListItem;
```

Description Read *Selected* to access the properties of the first selected item in the list. If *SelCount* is 0, *Selected* is **nil**. If *SelCount* is greater than 1, subsequent selected items can be located by checking the *Selected* property of the items found using the *GetNextItem* method.

Set the *Selected* property to select an item in the list. If *MultiSelect* is *True*, setting *Selected* adds an item to the selection. If *MultiSelect* is *False*, setting *Selected* changes the selection. Setting *Selected* to **nil** deselects all items in the list.

When an item is selected, the *OnChanging* and *OnChange* events are generated.

See also *GetNextItem* method, *MultiSelect* property, *OnChange* event, *OnChanging* event, *SelCount* property

ShowColumnHeaders

ShowColumnHeaders determines whether the column headers are displayed.

```
property ShowColumnHeaders: Boolean;
```

Description Set *ShowColumnHeaders* to specify whether the columns in the list view should display column headers. A column header is a panel with a caption that identifies the category of items in the column of the list view. Headers are useful for labeling columns of subitems that provide additional information in a list view. Use the *Columns* property to create or add headers.

Note To use columns in a list view, the *ViewStyle* property must be set to *vsReport*.

See also *ColumnClick* property, *Columns* property, *ViewStyle* property

SmallImages

SmallImages provides a list of icon images to display for each item in the list when *ViewStyle* is not *vsIcon*.

```
property SmallImages: TImageList;
```

Description Set *SmallImages* to specify the icons that should be displayed next to items in the list when *ViewStyle* is *vsSmallIcon*, *vsList*, or *vsReport*. Each item in the *Items* list can be associated with an icon in this image list by setting its *ImageIndex* property.

See also *Items* property, *LargeImages* property, *StateImages* property, *TImageList* object, *ViewStyle* property

C

SortType

SortType determines if and how the items in the list are automatically sorted.

```
type TSortType = (stNone, stData, stText, stBoth);

property SortType: TSortType;
```

Description Set *SortType* to specify the order of items in the list. If *SortType* is not *stNone*, the list items in the *Items* property are automatically sorted. Once a list view is sorted, the original order is lost. That is, setting the *SortType* back to *stNone* will not restore the original order of items. These are the possible values:

Value	Meaning
stNone	No sorting is done.
stData	Items are sorted based on the *Data* property of the *TListItem* objects. An *OnCompare* event handler must be provided to compare *Data* values.

Value	Meaning
stText	Items are sorted based on the *Caption* property of the *TListItem* objects.
stBoth	Items are sorted based on both the *Data* and *Caption* properties of the *TListItem* objects. The *Data* values are ignored unless an *OnCompare* event handler is provided.

See also *Items* property, *OnCompare* event, *TListItem.Caption* property, *TListItem.Data* property

StateImages

StateImages provides a list of additional icon images to display next to each item in the list view.

```
property StateImages: TImageList;
```

Description Set *StateImages* to specify the icons that should be displayed next to items in the list. Each item in the *Items* list can be associated with an icon in this image list by setting its *StateIndex* property.

See also *Items* property, *LargeImages* property, *SmallImages* property, *TImageList* object, *ViewStyle* property

TopItem

TopItem is the topmost visible item in the list view.

```
property TopItem: TListItem;
```

Description Read *TopItem* to determine the first item in the list view that the user can see. *TopItem* reflects how far the list view has been scrolled.

TopItem is only implemented when *ViewStyle* is *vsReport* or *vsList*. For *vsIcon* and *vsSmallIcon* view styles, use the *ViewOrigin* property to determine the scroll position.

See also *ViewOrigin* property, *ViewStyle* property

ViewOrigin

ViewOrigin returns the coordinates of the upper left corner of the list, relative to the visible area of the list view control.

```
property ViewOrigin: TPoint;
```

Description Read *ViewOrigin* to determine the logical origin of the list image. When the list view is not scrolled, *ViewOrigin* is (0,0). Otherwise, use the x coordinate to determine the number of pixels the view is scrolled horizontally, and the y coordinate to determine the number of pixels the view is scrolled vertically.

ViewOrigin is only implemented when *ViewStyle* is *vsIcon* or *vsSmallIcon*. For *vsReport* or *vsList*, use the *TopItem* property instead.

See also *TopItem* property, *ViewStyle* property

ViewStyle

ViewStyle determines how the list items are displayed.

property ViewStyle: TViewStyle;

Description Set *ViewStyle* to control the visual display of items in a list view. The items can be displayed as a set of movable icons, or as columns of text. These are the possible values:

Value	Meaning
vsIcon	Each item appears as a full-sized icon with a label below it. The user can drag the items to any location in the list view window.
vsSmallIcon	Each item appears as a small icon with the label to the right of it. The user can drag the items to any location.
vsList	Each item appears as a small icon with a label to the right of it. Items are arranged in columns and cannot be dragged by the user.
vsReport	Each item appears on its own line with information arranged in columns. The leftmost column contains the small icon and label, and subsequent columns contain subitems as specified by the application. Unless *ShowColumnHeaders* is *False*, each column has a header.

See also *IconOptions* property, *LargeImages* property, *ShowColumnHeaders* property, *SmallImages* property, *StateImages* property

VisibleRowCount

VisibleRowCount returns the number of items that can fit vertically in the visible area of a list view.

property VisibleRowCount: Integer;

Description Read *VisibleRowCount* to determine the number of items from a single column that can appear in the visible area of a list view when the *ViewStyle* is either *vsList* or *vsReport*. Only fully visible items are counted. If *VisibleRowCount* is less than the total number of items in a column, use the *TopItem* property to determine if the list view is scrolled.

See also *TopItem* property, *ViewStyle* property

Events

The following table lists the events of *TCustomListView*:

Event	Visibility	Documented in
OnChange	protected	TCustomListView
OnChanging	protected	TCustomListView

Event	Visibility	Documented in
OnColumnClick	protected	TCustomListView
OnCompare	protected	TCustomListView
OnDeletion	protected	TCustomListView
OnEdited	protected	TCustomListView
OnEditing	protected	TCustomListView
OnInsert	protected	TCustomListView

OnChange

OnChange occurs immediately after an item in the list changes.

```
type
  TItemChange = (ctText, ctImage, ctState);
  TLVChangeEvent = procedure(Sender: TObject; Item: TListItem; Change: TItemChange)
    of object;

property OnChange: TLVChangeEvent;
```

Description Write an *OnChange* event handler to respond to changes in the list items. *OnChange* allows a response once the list has been successfully changed.

The *Item* parameter is the list item that just changed. The *Change* parameter indicates the type of change that just occurred. *Change* is *ctText* if the *Caption* property of the item changed. *Change* is *ctImage* if the *ImageIndex* property of the item changed or the appropriate image list changed in the list view. *Change* is *ctState* if the *Cut, Focused,* or *Selected* property of the item changed.

See also *LargeImages* property, *OnChanging* event, *SmallImages* property, *TListItem* object

OnChanging

OnChanging occurs when an item in the list is about to change.

```
type
  TItemChange = (ctText, ctImage, ctState);
  TLVChangingEvent = procedure(Sender: TObject; Item: TListItem; Change: TItemChange;
    var AllowChange: Boolean) of object;

property OnChanging: TLVChangingEvent;
```

Description Write an *OnChanging* event handler to conditionally block changes to an item in the list. Set the *AllowChange* parameter to *False* to prevent the change from taking place. The *Item* parameter is the list item that is about to change. The *Change* parameter indicates the type of change that is about to happen.

Change is *ctText* if the *Caption* property of the item is about to change. *Change* is *ctImage* if the *ImageIndex* property of the item is about to change. *Change* is *ctState* if the *Cut, Focused,* or *Selected* property of the item is about to change.

See also *OnChange* event, *TListItem* object

OnColumnClick

OnColumnClick occurs when the user clicks on a column header in a list view.

```
type TLVColumnClickEvent = procedure(Sender: TObject; Column: TListColumn) of object;

property OnColumnClick: TLVColumnClickEvent;
```

Description Write an *OnColumnClick* event handler to respond to mouse clicks made in the column header when *ViewStyle* is *vsReport*. The Column parameter is the *TListColumn* object from the *Columns* property that describes the column that was clicked.

OnColumnClick only occurs when *ViewStyle* is *vsReport*, *ShowColumnHeaders* is *True*, and *ColumnClick* is *True*.

See also *ColumnClick* property, *Columns* property, *ShowColumnHeaders* property, *TListColumn* object, *ViewStyle* property

OnCompare

OnCompare occurs to when two items need to be compared during a sort of the list.

```
type TLVCompareEvent = procedure(Sender: TObject; Item1, Item2: TListItem; Data: Integer;
  var Compare: Integer) of object;

property OnCompare: TLVCompareEvent;
```

Description Write an *OnCompare* event handler to implement a sort order for the list. An *OnCompare* event handler is called when the *SortType* property is *stData* or *stBoth*, when the *AlphaSort* method is called, or when the *CustomSort* method is called without a *SortProc* parameter.

The *OnCompare* event handler compares the list items passed as the *Item1* and *Item2* parameters. If *Item1* is the same as *Item2* in the sort order, set the *Compare* parameter to 0. If *Item1* is less than *Item2*, set the *Compare* parameter to a value less than 0. If *Item1* is greater than *Item2*, set the *Compare* parameter to a value greater than 0. The *Data* parameter is 0 when the event handler is called to maintain the sort order of a list view with a *SortType* of *stData* or *stBoth*. Similarly, when *OnCompare* occurs in response to the *AlphaSort* method, the *Data* parameter is 0. When *OnCompare* occurs in response to the *CustomSort* method, the *Data* parameter is the value of the *LParam* parameter of *CustomSort*.

See also *AlphaSort* method, *CustomSort* method, *SortType* property

OnDeletion

OnDeletion occurs when an item in the list view is about to be deleted.

```
type TLVDeletedEvent = procedure(Sender: TObject; Item: TListItem) of object;
property OnDeletion: TLVDeletedEvent;
```

Description Write an *OnDeletion* event handler to respond when an item in the list is about to be deleted. The *Item* parameter is the *TListItem* object from the *Items* property that is about to be deleted.

See also *Items* property, *OnInsert* event, *TListItem* object

OnEdited

OnEdited occurs immediately after the user edits an item's *Caption* property.

```
type TLVEditedEvent = procedure(Sender: TObject; Item: TListItem; var S: string) of object;
property OnEdited: TLVEditedEvent;
```

Description Write an *OnEdited* event to take specific action when a user has finished editing the text of a list item. An *OnEdited* event handler can respond to the value the user typed, or change the value that is assigned to the list item's *Caption* property. For example, if an *OnEdited* event handler can check the validity of the string typed by the user, and only allow the *Caption* property to be changed if the new value is valid.

The *Item* parameter is the list item that was just edited. The *S* parameter is the new value the user typed for the item's *Caption* property. Change the value of *S* to override the value the user typed. When *OnEdited* occurs, the item's *Caption* property has not yet been changed, so *S* can still be compared to the original value of *Caption*.

This event can occur only if *ReadOnly* is set to *False*.

See also *OnEditing* event, *ReadOnly* property, *TListItem.Caption* property

OnEditing

OnEditing occurs when the user starts to edit a list item's *Caption* property.

```
type TLVEditingEvent = procedure(Sender: TObject; Item: TListItem; var AllowEdit: Boolean)
  of object;
property OnEditing: TLVEditingEvent;
```

Description Write an *OnEditing* event handler to take specific action just before a user edits a list item's *Caption* property. To prevent the user from editing the item specified by the *ListItem* parameter, set the *AllowEdit* parameter to *False*.

This event can occur only if the *OnEditing*'s *ReadOnly* property is set to *False*.

See also *OnEdited* event, *ReadOnly* property, *TListItem.Caption* property

OnInsert

OnInsert occurs immediately after a new item is inserted into the list view.

```
type TLVDeletedEvent = procedure(Sender: TObject; Item: TListItem) of object;
property OnInsert: TLVDeletedEvent;
```

Description Write an *OnInsert* event handler to respond when an item has just been added to the list. The *Item* parameter is the *TListItem* object from that was added to the *Items* property.

See also *Items* property, *OnDeletion* event, *TListItem* object

Methods

The following table lists the methods of *TCustomListView*:

Method	Visibility	Documented in
AlphaSort	public	TCustomListView
Arrange	public	TCustomListView
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanChange	protected	TCustomListView
CanEdit	protected	TCustomListView
CanFocus	public	TWinControl
Change	protected	TCustomListView
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ColClick	protected	TCustomListView
ColumnsShowing	protected	TCustomListView
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomListView
CreateListItem	protected	TCustomListView
CreateParams	protected	TCustomListView
CreateParented	public	TWinControl
CreateWnd	protected	TCustomListView
CustomSort	public	TCustomListView
DefaultHandler	public	TObject
Delete	protected	TCustomListView
Destroy	public	TCustomListView

Method	Visibility	Documented in
DestroyComponents	public	TComponent
Destroying	public	TComponent
DestroyWnd	protected	TCustomListView
DisableAlign	public	TWinControl
Dispatch	public	TObject
DoEndDrag	protected	TCustomListView
DoStartDrag	protected	TCustomListView
DragDrop	public	TControl
Dragging	public	TControl
Edit	protected	TCustomListView
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindCaption	public	TCustomListView
FindComponent	public	TComponent
FindData	public	TCustomListView
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetDragImages	protected	TCustomListView
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetItemAt	public	TCustomListView
GetItemIndex	protected	TCustomListView
GetNamePath	public	TPersistent
GetNearestItem	public	TCustomListView
GetNextItem	public	TCustomListView
GetParentComponent	public	TComponent
GetSearchString	public	TCustomListView
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent

Method	Visibility	Documented in
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
IsEditing	public	TCustomListView
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TCustomListView
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
Scroll	public	TCustomListView
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
StringWidth	public	TCustomListView
Update	public	TWinControl
UpdateColumn	protected	TCustomListView
UpdateColumns	protected	TCustomListView
UpdateControlState	public	TWinControl
UpdateItems	public	TCustomListView
WndProc	protected	TCustomListView

C

AlphaSort

AlphaSort sorts the items in the list view.

```
function AlphaSort: Boolean;
```

Description Call *AlphaSort* to sort the items in the manner maintained automatically by when the *SortType* property is not *stNone*. If an *OnCompare* event handler is assigned, *AlphaSort* uses that event handler to define the sort order.

Otherwise, *AlphaSort* sorts all the list items alphabetically in ascending order. If successful, the method returns *True*.

See also *CustomSort* method, *OnCompare* event, *SortType* property

Arrange

Arrange arranges icons in a list view that has a *ViewStyle* of *vslcon* or *vsSmallIcon*.

```
type TListArrangement = (arAlignBottom, arAlignLeft, arAlignRight, arAlignTop, arDefault,
  arSnapToGrid);
procedure Arrange(Code: TListArrangement);
```

Description Call *Arrange* to clean up the arrangement of the icons when the *ViewStyle* property is *vslcon* or *vsSmallIcon*. The *Code* parameter specifies how to arrange the icons. These are the possible values for the *Code* parameter.

Value	Meaning
arAlignBottom	Aligns items along the bottom edge of the list view.
arAlignLeft	Aligns items along the left edge of the window.
arAlignRight	Aligns items along the right edge of the window.
arAlignTop	Aligns items along the top edge of the window.
arDefault	Aligns items according to the style associated with the list view when the window was created. For *TCustomListView*, this is aligned along the top.
arSnapToGrid	Snaps all icons to the nearest grid position.

See also *ViewStyle* property

CanChange

CanChange specifies whether the properties of a list item can be changed.

```
function CanChange(Item: TListItem; Change: Integer): Boolean; dynamic;
```

Description *CanChange* is called automatically when an attempt is made to change the item specified by the *Item* parameter. The *Change* parameter is a Windows-defined constant that specifies the type of change that was attempted. *CanChange* generates an *OnChanging* event to determine whether the change should be permitted. *CanChange* returns *True* if the change can proceed. *CanChange* returns *False* if the change should be blocked.*CanChange* always returns *True* if there is no *OnChanging* event handler.

Override *CanChange* to add additional logic to determine whether a change should proceed or to block the *OnChanging* event.

See also *OnChanging* event

CanEdit

CanEdit specifies whether the text of a list item can be changed by the user.

```
function CanEdit(Item: TListItem): Boolean; dynamic;
```

Description *CanEdit* is called automatically when the user attempts to edit the text of the list item specified by the *Item* parameter. *CanEdit* generates an *OnEditing* event to determine whether to allow the user to edit the item. *CanEdit* returns *True* if the user can proceed. *CanEdit* returns *False* if the user should be blocked from editing the item.*CanEdit* always returns *True* if there is no *OnEditing* event handler.

Override *CanEdit* to add additional logic to determine whether an item can be edited or to block the *OnEditing* event.

See also *OnEditing* event

Change

Change generates an *OnChange* event.

```
procedure Change(Item: TListItem; Change: Integer); dynamic;
```

Description *Change* is called automatically after the properties of the item specified by the *Item* parameter change. The *Change* parameter is a Windows-defined constant that specifies the type of change that occurred.

Override *Change* to perform other actions when the list items change or to block the *OnChange* event.

See also *OnChange* event

C

ColClick

ColClick generate an *OnColumnClick* event.

```
procedure ColClick(Column: TListColumn); dynamic;
```

Description *ColClick* is called automatically when the user clicks on a column header if the *ColumnClick* and *ShowColumnHeaders* properties are *True* and the *ViewStyle* property is *vsReport*. Override *ColClick* to take additional action when the user clicks on a column header or to conditionally block the *OnColumnClick* event.

See also *ColumnClick* property, *OnColumnClick* event, *ShowColumnHeaders* property, *ViewStyle* property

ColumnsShowing

ColumnsShowing indicates whether the *ViewStyle* property is *vsReport*.

```
function ColumnsShowing: Boolean;
```

Description Call *ColumnsShowing* to determine whether the list view displays its items in columns. *ColumnsShowing* returns *True* if the value of *ViewStyle* is *vsReport*, *False* otherwise.

See also *ViewStyle* property

Create

Create creates and initializes an instance of *TCustomListView*.

```
constructor Create(AOwner: TListItems);
```

Description Call *Create* to instantiate a list view at runtime. List view controls placed on forms at design time are created automatically.

After calling the inherited constructor, *Create* initializes the following properties:

- *BorderStyle* to *bsSingle*.
- *ColumnClick* and *ShowColumnHeaders* to *True*.
- *ControlStyle* to [*csClickEvents*, *csDoubleClick*, *csDisplayDragImage*, *csSetCaption*].
- *Height* to 150 and *Width* to 250.
- *HideSelection* to *True*.
- *TabStop* to *True*.
- *ViewStyle* to *vsIcon*.

See also *BorderStyle* property, *ColumnClick* property, *ControlStyle* property, *Height* property, *HideSelection* property, *ShowColumnHeaders* property, *TabStop* property, *TWinControl.Create* method, *ViewStyle* property, *Width* property

CreateListItem

CreateListItem creates a list item for the collection specified by the *Items* property.

```
function CreateListItem: TListItem; virtual;
```

Description Call *CreateListItem* to create a *TListItem* object that will be added to the *Items* collection. *CreateListItem* creates the list item but does not insert it into the collection.

See also *Items* property, *TListItem* object

CreateParams

CreateParams initializes the window-creation parameter record passed in the *Params* parameter.

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description *CreateParams* is called automatically to initialize the window-creation parameters when the list view needs to create its window. *CreateParams* overrides the inherited method to specify a list view window that implements the property settings of the list view control.

When overriding the *CreateParams* method, be sure to call the inherited method before changing any of the values in the *Params* parameter.

See also *TWinControl.CreateParams* method

CreateWnd

CreateWnd creates the window used by the list view.

```
procedure CreateWnd; override;
```

Description *CreateWnd* creates the list view's window, based on the parameter values provided by the *CreateParams* method. *CreateWnd* then sends messages to the newly-created window so that it reflects the current property settings of the list view object.

See also *CreateParams* method

CustomSort

CustomSort method sorts the items in the list using the specified ordering function.

```
type TLVCompare = function(lParam1, lParam2, lParamSort: Integer): Integer stdcall;

function CustomSort(SortProc: TLVCompare; lParam: Longint): Boolean;
```

Description Call *CustomSort* to sort the items in the list using the ordering function defined by the *SortProc* parameter. The *SortProc* parameter specifies an ordering function that compares the list items passed as lParam1 and lParam2. The ordering function returns an integer that indicates whether lParam1 is the same as lParam2 (return value is 0), lParam1 is greater than lParam2 (return value is greater than 0), or lParam1 is less than lParam2 (return value is less than 0). The *lParam* parameter of *CustomSort* is an optional value that is passed as the third parameter to the ordering function.

If the *SortProc* parameter is **nil**, *CustomSort* compares the list items by generating an *OnCompare* event. This allows the *OnCompare* event handler to provide different ordering criteria (such as ascending vs. descending order), based on the value of the *lParam* parameter.

If the ordering function is not provided and there is no *OnCompare* event handler, *CustomSort* sorts items alphabetically by their *Caption* values.

CustomSort returns *True* if the list is successfully sorted.

See also *AlphaSort* method, *OnCompare* event

Delete

Delete removes an item from the list view and frees the associated memory.

```
procedure Delete(Item: TListItem); dynamic;
```

Description Call *Delete* to remove an item from the list view. *Delete* generates an *OnDeletion* event before freeing the list item.

See also *OnDeletion* event

Destroy

Destroy destroys an instance of *TCustomListView*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly. Instead, call *Free*, which verifies that the list item is not already freed before calling *Destroy*.

See also *Free* method

DestroyWnd

DestroyWnd destroys the window referenced by the list view's window handle.

```
procedure DestroyWnd; override;
```

Description Before the window is destroyed, *DestroyWnd* saves a copy of the list view in internal storage. The inherited method then frees any device contexts, and destroys the window.

See also *CreateWnd* method, *TWinControl.DestroyWnd* method

DoEndDrag

DoEndDrag generates an *OnEndDrag* event.

```
procedure DoEndDrag(Target: TObject; X, Y: Integer); override;
```

Description After generating an *OnEndDrag* event, *DoEndDrag* clears the internal variables used to track which item is the drop target.

See also *DoStartDrag* method, *DropTarget* property, *TControl.OnEndDrag* event

DoStartDrag

DoStartDrag generates an *OnStartDrag* event and begins tracking the drop target.

```
procedure DoStartDrag(var DragObject: TDragObject); override;
```

Description *DoStartDrag* is called automatically when the user starts dragging a list item with the mouse. After generating an *OnStartDrag* event, *DoStartDrag* generates a drag image and begins calculating drop target positions.

See also *DoEndDrag* method, *DropTarget* property, *TControl.OnStartDrag* event

Edit

Edit sets the *Caption* property of a list item after it has been edited by the user.

```
procedure Edit(const Item: TLVItem); dynamic;
```

Description *Edit* is called automatically when the user finishes editing the label of a list item. *Edit* obtains the newly edited value from the Windows list item and generates an *OnEdited* event. After allowing the *OnEdited* event handler to change the value supplied by the user, *Edit* sets the *Caption* property of the list item.

See also *OnEdited* event, *TListItem.Caption* property

FindCaption

FindCaption returns a list view item, if any, with the specified caption.

```
function FindCaption(StartIndex: Integer; Value: string; Partial, Inclusive, Wrap: Boolean):
  TListItem;
```

Description Call *FindCaption* to search for a list view item labelled by the string specified as the *Value* parameter. If the *Inclusive* parameter is *True*, the list item specified by *StartIndex* is the first item checked. Otherwise, the search starts on the next item. If the *Wrap* parameter is *True*, the search continues at the top of the list if a match has not been found before reaching the bottom. *FindCaption* returns the first item with the *Caption* property equal to *Value*. If the *Partial* parameter is *True*, a *Caption* matches if it starts with the substring *Value*. If *Partial* is *False*, the *Caption* must match *Value* exactly. If no item is found, *FindCaption* returns **nil**.

See also *FindData* method, *TListItem.Caption* property

FindData

FindData returns a list view item, if any, associated with the specified data.

```
function FindData(StartIndex: Integer; Value: Pointer; Inclusive, Wrap: Boolean): TListItem;
```

Description Call *FindData* to locate a list view item with its *Data* property equal to the *Value* parameter. If the *Inclusive* parameter is *True*, the list item specified by *StartIndex* is the first item checked. Otherwise, the search starts on the next item. If the *Wrap* parameter is *True*, the search continues at the top of the list if a match has not been found before reaching the bottom. *FindData* returns the first item with a *Data* property equal to the *Value* parameter. If no item is found, *FindData* returns **nil**.

See also *FindCaption* method, *TListItem.Data* property

GetDragImages

GetDragImages returns a drag cursor if a list item is being dragged.

```
function GetDragImages: TCustomImageList; override;
```

Description *GetDragImages* is called automatically to obtain a list of drag images when the user drags the mouse with the button held down. If the user is dragging a list item, *GetDragImages* returns the drag image list that is created when the user starts dragging a list item. If the user did not start dragging with the mouse over a list item, *GetDragImages* returns **nil**.

See also *DoStartDrag* method

GetItemAt

GetItemAt returns the list item at the specified position.

```
function GetItemAt(X, Y: Integer): TListItem;
```

Description Call *GetItemAt* to obtain the *TListItem* object for the list item found at the position specified by the *X* and *Y* parameters. *X* and *Y* specify the coordinates of the position, in pixels, relative to the top left corner of the list view. If there is no item at the location, *GetItemAt* returns **nil**.

See also *GetNearestItem* method, *Items* property

GetItemIndex

GetItemIndex returns the index of the specified item in the *Items* property array.

```
function GetItemIndex(Value: TListItem): Integer;
```

Description Call *GetItemIndex* to determine the position of a list item, where 0 is the index of the first item, 1 is the index of the second item, and so on. If the item specified by the *Value* property is not in the list view, *GetItemIndex* returns –1.

See also *Items* property

GetNearestItem

GetNearestItem returns the list item nearest a specified point, in a particular direction.

```
function GetNearestItem(Point: TPoint; Direction: TSearchDirection): TListItem;
```

Description Call *GetNearestItem* to locate a list item from a position specified in pixel coordinates relative to the top left corner of the list view. *GetNearestItem*

starts looking at the position specified by the *Point* parameter, and moves in the direction indicated by the *Direction* parameter until it locates a list item.

The *Direction* parameter can be one of the following values:

Value	Meaning
sdLeft	*GetNearestItem* looks to the left of the specified point.
sdRight	*GetNearestItem* looks to the right of the specified point.
sdAbove	*GetNearestItem* looks above the specified point.
sdBelow	*GetNearestItem* looks below the specified point.
sdAll	*GetNearestItem* returns the closest item in any direction.

If no item is found, *GetNearestItem* returns **nil**.

See also *GetItemAt* method, *GetNextItem* method, *Items* property

GetNextItem

GetNextItem returns the next list item after the *StartItem* in the specified direction.

```
function GetNextItem(StartItem: TListItem; Direction: TSearchDirection; States: TItemStates) :
  TListItem;
```

Description Call *GetNextItem* to find the next list item after *StartItem* in the direction given by the *Direction* parameter. Only items in the state indicated by the *States* parameter are considered.
The *Direction* parameter can be one of the following values:

Value	Meaning
sdLeft	*GetNextItem* looks to the left of the specified item.
sdRight	*GetNextItem* looks to the right of the specified item.
sdAbove	*GetNextItem* looks above the specified item.
sdBelow	*GetNextItem* looks below the specified item.
sdAll	*GetNextItem* looks in the order that items appear in the *Items* property.

The State parameter can be one of the following values:

Value	Meaning
isNone	Only items in the default state are returned
isCut	Only items marked for a cut and paste operation are returned.
isDropHilited	Only items highlighted as a drag-and-drop target are returned
isFocused	The value of the *ItemFocused* property is returned.
isSelected	Only selected items are returned.

See also *DropTarget* property, *GetNearestItem* method, *ItemFocused* property, *Selected* property

GetSearchString

GetSearchString returns the incremental search string of a list view control.

```
function GetSearchString: string;
```

Description Call *GetSearchString* to obtain the character sequence typed by the user while the list view has input focus. Each time the user types a character, the list view appends the character to the search string and searches for a matching item. A timeout period of less than a second is associated with each character that the user types. If the timeout period elapses before the user types another character, the incremental search string is reset. If the system finds a match, it selects the item and, if necessary, scrolls it into view.

See also *Selected* property, *TListItem.Caption* property

IsEditing

IsEditing returns *True* if any item in the list view is being edited.

```
function IsEditing: Boolean;
```

Description Call *IsEditing* to determine whether the user is editing the caption of any of the items in the list.

See also *OnEdited* event, *OnEditing* event, *TListItem.Caption* property

Notification

Notification updates an image property to **nil** if the image list is freed.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description After calling the inherited method, *Notification* checks whether the image list specified by the *LargeImages*, *SmallImages*, or *StateImages* property is about to be destroyed. If so, *Notification* sets the appropriate property to **nil**.

See also *LargeImages* property, *SmallImages* property, *StateImages* property

Scroll

Scroll scrolls the content of a list view by the indicated amount.

```
procedure Scroll(DX, DY: Integer);
```

Description Call *Scroll* to scroll the content of a list view by *DX* pixels horizontally and *DY* pixels vertically.

See also *TopItem* property, *ViewOrigin* property

StringWidth

StringWidth returns the width of a string, using the list view's current font.

```
function StringWidth(S: string): Integer;
```

Description Call *StringWidth* to determine the space, in pixels, required to display a string using the list view's current font.

See also *TControl.Font* property

UpdateColumn

UpdateColumn updates the image of the column indicated by *Index*.

```
procedure UpdateColumn(Index: Integer);
```

Description *UpdateColumn* is called automatically when the display properties of a column in the list view change. *UpdateColumn* adjusts the image of the column to match the properties of the *TListColumn* object indexed by the *Index* parameter in the *Columns* property.

UpdateColumn does nothing if *ColumnsShowing* returns *False*.

See also *Columns* property, *ColumnsShowing* method, *TListColumn* object, *UpdateColumns* method

UpdateColumns

UpdateColumns causes the list view control to update the image of all columns.

```
procedure UpdateColumns;
```

Description *UpdateColumns* is called automatically when the properties of several columns in the list view change. *UpdateColumn* adjusts the image of every column to match the properties of the corresponding *TListColumn* object listed in the *Columns* property.

UpdateColumns does nothing if *ColumnShowing* returns *False*.

See also *Columns* property, *ColumnShowing* method, *TListColumn* object, *UpdateColumn* method

UpdateItems

UpdateItems redraws a range of items, starting at *FirstIndex* and ending at *LastIndex*.

```
procedure UpdateItems(FirstIndex, LastIndex: Integer);
```

Description Call *UpdateItems* after making changes that affect the display image of the list items in the specified range.

See also *Items* property

WndProc

WndProc provides specific message responses for the list view.

procedure WndProc(**var Message**: TMessage); **override**;

Description *WndProc* overrides the inherited method to prevent the list view from redrawing while the items are being updated.

See also *TWinControl.WndProc* method

TCustomMaskEdit

Unit: Mask

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomEdit • TCustomMaskEdit

TCustomMaskEdit is the abstract base type for edit controls that provide a mask for checking the validity of text typed into the control.

Description *TCustomMaskEdit* introduces new properties and methods to expand the capabilities of *TCustomEdit* for validity checking using an edit mask. To an edit control that takes advantage of the masking capabilities of *TCustomMaskEdit*, use a *TMaskEdit* object. To create an edit control that does not validate the input, use the *TEdit* object instead.

Do not create instances of *TCustomMaskEdit*. Use *TCustomMaskEdit* as a base class when declaring control objects that implement a masked edit control. Properties and methods of *TCustomMaskEdit* provide basic behavior that descendent classes inherit as well as behavior that components can override to customize their behavior.

See also *EDBEditError* object, *TDBEdit* object, *TEdit* object, *TMaskEdit* object

Properties

The following table lists the properties of *TCustomMaskEdit*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
EditMask	protected		TCustomMaskEdit
EditText	public		TCustomMaskEdit
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
IsMasked	public, read-only		TCustomMaskEdit
Left	published		TControl
MaskState	protected		TCustomMaskEdit
MaxLength	protected	0	TCustomMaskEdit
Modified	public		TCustomEdit
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
SelLength	public		TCustomEdit
SelStart	public		TCustomEdit
SelText	public		TCustomEdit
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	public		TCustomMaskEdit
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

EditMask

EditMask is the mask that represents what text is valid for the masked edit control.

property EditMask: string;

Description

Use *EditMask* to restrict the characters a user can enter into the masked edit control to valid characters and formats. If the user attempts to enter an invalid character, the edit control does not accept the character. Validation is performed on a character-by-character basis by the *ValidateEdit* method.

A mask consists of three fields with semicolons separating the fields. The first part of the mask is the mask itself. The second part is the character that determines whether the literal characters of a mask are saved as part of the data. The third part of the mask is the character used to represent unentered characters in the mask.

These are the special characters used in the first field of the mask:

Character	Meaning in mask
!	If a ! character appears in the mask, optional characters are represented in the *EditText* as leading blanks. If a ! character is not present, optional characters are represented in the *EditText* as trailing blanks.
>	If a > character appears in the mask, all characters that follow are in uppercase until the end of the mask or until a < character is encountered.
<	If a < character appears in the mask, all characters that follow are in lowercase until the end of the mask or until a > character is encountered.
<>	If these two characters appear together in a mask, no case checking is done and the data is formatted with the case the user uses to enter the data.
\	The character that follows a \ character is a literal character. Use this character to use any of the mask special characters as a literal in the data.
L	The L character requires an alphabetic character only in this position. For the US, this is A-Z, a-z.
l	The l character permits only an alphabetic character in this position, but doesn't require it.
A	The A character requires an alphanumeric character only in this position. For the US, this is A-Z, a-z, 0-9.
a	The a character permits an alphanumeric character in this position, but doesn't require it.
C	The C character requires an arbitrary character in this position.
c	The c character permits an arbitrary character in this position, but doesn't require it.
0	The 0 character requires a numeric character only in this position.
9	The 9 character permits a numeric character in this position, but doesn't require it.
#	The # character permits a numeric character or a plus or minus sign in this position, but doesn't require it.

Character	Meaning in mask
:	The : character is used to separate hours, minutes, and seconds in times. If the character that separates hours, minutes, and seconds is different in the regional settings of the Control Panel utility on your computer system, that character is used instead.
/	The / character is used to separate months, days, and years in dates. If the character that separates months, days, and years is different in the regional settings of the Control Panel utility on your computer system, that character is used instead.
;	The ; character is used to separate the three fields of the mask.
_	The _ character automatically inserts spaces into the text. When the user enters characters in the field, the cursor skips the _ character.

Any character that does not appear in the preceding table can appear in the first part of the mask as a literal character. Literal characters must be matched exactly in the edit control. They are inserted automatically, and the cursor skips over them during editing. The special mask characters can also appear as literal characters if preceded by a backslash character (\).

The second field of the mask is a single character that indicates whether literal characters from the mask should be included as part of the text for the edit control. For example, the mask for a telephone number with area code could be the following string:

(000)_000-0000;0;*

The 0 in the second field indicates that the *Text* property for the edit control would consist of the 10 digits that were entered, rather than the 14 characters that make up the telephone number as it appears in the edit control.

A 0 in the second field indicates that literals should not be included, any other character indicates that they should be included. The character to indicate whether literals should be included can be changed in the *EditMask* property editor, or programmatically by changing the *MaskNoSave* typed constant.

The third field of the mask is the character that appears in the edit control for blanks (characters that have not been entered). By default, this is the same as the character that stands for literal spaces. The two characters appear the same in an edit window. However, when a user edits the text in a masked edit control, the cursor selects each blank character in turn, and skips over the space character.

Setting *EditMask* to an empty string removes the mask.

Note When working with multibyte character sets, such as Japanese shift-JIS, each special mask character represents a single byte. To specify double-byte characters using the L,l,A,a,C, or c specifiers, the mask characters must be doubled as well. For example, LL would represent two single-byte alphabetic characters or a one double-byte character. Only single-byte literal characters are supported.

See also *EditText* property, *IsMasked* property, *MaskState* property, *Text* property, *ValidateEdit* method, *ValidateError* method

EditText

EditText is the text that appears in the window of the masked edit control.

```
property EditText: string;
```

Description *EditText* is the value of the text for the edit control after it is formatted using the *EditMask*. Before the text is fully entered into a masked edit control, the *EditText* includes a blank character for each unentered character. As characters are entered, the blank characters in the *EditText* are replaced by the values that the user types.

If literal mask characters are included in the text, and no character is substituted for blanks, the values of *EditText* and *Text* are the same.

Note *EditText* only applies at runtime.

See also *EditMask* property, *Text* property

IsMasked

IsMasked indicates whether a mask exists for the edit control.

```
property IsMasked: Boolean;
```

Description Read *IsMasked* to determine whether the *EditMask* property has been set. If *IsMasked* is *True*, the control has a mask, and input is validated at runtime. If *IsMasked* is *False*, the control does not restrict editing at runtime.

See also *EditMask* property

MaskState

MaskState indicates state information about how the mask is being applied.

```
TMaskedState = set of (msMasked, msReEnter, msDBSetText);

property MaskState: TMaskedState;
```

Description *MaskState* is used internally to determine appropriate actions based on how the mask is currently being applied. *MaskState* is a set drawn from among the following values:

Value	Meaning
msMasked	A mask has been assigned. *MaskState* includes *msMasked* when *IsMasked* is *True*.
msReEnter	An invalid character has been rejected and the next edit should be a valid replacement character.
msDBSetText	The text comes from a database field and should not be validated.

MaskState is exposed as a protected property so that derived classes can maintain and use this state information in the implementation of properties and methods.

See also *IsMasked* property, *ValidateEdit* method

MaxLength

MaxLength specifies the maximum number of characters that can appear in the edit control.

```
property MaxLength: Integer;
```

Description *MaxLength* is the length of the *EditText*. Set *MaxLength* to limit the number of characters that can appear in the edit control when there is no *EditMask*. If there is an *EditMask*, *MaxLength* is implied by the mask itself, and cannot be changed. The *EditText* contains blank characters for each character that has not been entered, padding the end or beginning of every optional section, so it remains constant in length.

See also *EditMask* property, *EditText* property, *IsMasked* property, *Text* property

Text

Text is the underlying text for the masked edit control before the mask has been applied.

```
property Text: string;
```

Description Use *Text* to determine the underlying value of the text before it has been formatted by the mask. *Text* may differ from the *EditText* visible in the masked edit control if the mask specifies that literal characters should be removed, if the mask includes spaces, or if the mask includes characters that have not yet been filled in by the user. *Text* will not contain the blank characters, spaces will not be replaced by _, and literal characters in the mask will be removed if the mask indicates they should be.

When setting *Text*, the value of the text is formatted using the mask, and the resulting *EditText* is used to update the window.

If there is no mask, *Text* is the text that appears in the edit control.

See also *EditMask* property, *EditText* property, *GetTextLen* method, *IsMasked* property, *ReformatText* method

Methods

The following table lists the methods of *TCustomMaskEdit*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
CheckCursor	protected	TCustomMaskEdit
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomMaskEdit
ClearSelection	public	TCustomEdit
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
CopyToClipboard	public	TCustomEdit
Create	public	TCustomMaskEdit
CreateParented	public	TWinControl
CutToClipboard	public	TCustomEdit
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EditCanModify	protected	TCustomMaskEdit
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent

Method	Visibility	Documented in
FreeOnRelease	public	TComponent
GetFirstEditChar	protected	TCustomMaskEdit
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetLastEditChar	protected	TCustomMaskEdit
GetMaxChars	protected	TCustomMaskEdit
GetNamePath	public	TPersistent
GetNextEditChar	protected	TCustomMaskEdit
GetParentComponent	public	TComponent
GetPriorEditChar	protected	TCustomMaskEdit
GetSel	protected	TCustomMaskEdit
GetSelTextBuf	public	TCustomEdit
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TCustomMaskEdit
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
KeyDown	protected	TCustomMaskEdit
KeyPress	protected	TCustomMaskEdit
KeyUp	protected	TCustomMaskEdit
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
PasteFromClipboard	public	TCustomEdit
Perform	public	TControl
Realign	public	TWinControl
ReformatText	protected	TCustomMaskEdit
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
Reset	protected	TCustomMaskEdit

Method	Visibility	Documented in
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectAll	public	TCustomEdit
SendToBack	public	TControl
SetBounds	public	TWinControl
SetCursor	protected	TCustomMaskEdit
SetFocus	public	TWinControl
SetSel	protected	TCustomMaskEdit
SetSelTextBuf	public	TCustomEdit
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl
ValidateEdit	public	TCustomMaskEdit
ValidateError	protected	TCustomMaskEdit

CheckCursor

CheckCursor moves the cursor to the closest editable character and selects it if nothing is currently selected.

```
procedure CheckCursor;
```

Description *CheckCursor* is called whenever the position of the cursor may have changed. It ensures that at least one character is always selected. Because editing a masked edit control consists of replacing the blank characters in the mask, there must always be a selection to replace. The selection corresponds to one or more nonliteral characters in the mask, which are either blank or contain characters for the user to re-enter.

See also *GetSel* method, *SetCursor* method, *SetSel* method

Clear

Clear sets the *Text* property to an empty string.

```
procedure Clear;
```

Description Use *Clear* to remove all text that was entered into the masked edit control. *Clear* sets the value of *Text* to an empty string. It does not clear the *EditMask*. If the edit control has a mask, *EditText* will not be empty, but will have all non-literal characters replaced by blanks.

See also *EditMask* property, *EditText* property, *IsMasked* property, *Reset* method, *Text* property

Create

Create creates an instance of *TCustomMaskEdit*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a *TCustomMaskEdit* object. *Create* initializes the *MaskState* and blank substitution character after calling the inherited constructor.

See also *TCustomEdit.Create* method

EditCanModify

EditCanModify permits the user to edit the *Text*.

function EditCanModify: Boolean; **virtual**;

Description The *EditCanModify* method defined by *TCustomMaskEdit* always returns *True*, allowing the *Text* to be edited. *EditCanModify* exists primarily to provide an interface for a method that can conditionally disallow changes to the *Text* by returning *False*.

See also *Text* property

GetFirstEditChar

GetFirstEditChar returns the index in the *EditMask* of the first nonliteral character, where zero is the first character.

function GetFirstEditChar: Integer;

Description Use *GetFirstEditChar* to find the first position of the *EditMask* where the user can modify the *EditText*. If there is no mask, *GetFirstEditChar* returns zero.

Use *GetFirstEditChar* as a starting point for programmatically processing all the nonliteral characters in the *EditText*. Use *GetNextEditChar* to continue to loop through the editable characters in the *EditText*, and *GetLastEditChar* to determine when all nonliteral characters have been processed.

See also *EditMask* property, *EditText* property, *GetLastEditChar* method, *GetNextEditChar* method, *GetPriorEditChar* method

GetLastEditChar

GetLastEditChar returns the index in the *EditMask* of the last nonliteral character, where zero is the first character.

function GetLastEditChar: Integer;

Description Use *GetLastEditChar* to find the last position of the *EditMask* where the user can modify the *EditText*. If there is no mask, *GetLastEditChar* returns the length of the text.

See also *EditMask* property, *EditText* property, *GetFirstEditChar* method, *GetMaxChars* property, *GetNextEditChar* method, *GetPriorEditChar* method

GetMaxChars

GetMaxChars returns the length of the *EditText*.

```
function GetMaxChars: Integer;
```

Description If there is a mask, *GetMaxChars* returns the length of the *EditText*, including all blanks. This is the same as *MaxLength*, because editing in a masked edit control consists of replacing nonliteral characters. If there is no mask, *GetMaxChars* returns the length of the text in the window. In this case, the value returned by *GetMaxChars* may be less than *MaxLength*, which is the maximum number of characters that can be entered into the edit control.

See also *EditText* property, *GetLastEditChar* method, *MaxLength* property, *Text* property

GetNextEditChar

GetNextEditChar returns the first index of a non-literal character equal to or greater than *Offset*, where zero is the first character.

```
function GetNextEditChar(Offset: Integer): Integer;
```

Description Call *GetNextEditChar* to find the first position of the *EditMask* at or following *Offset* where the user can modify the *EditText*. If there is no nonliteral character at or following *Offset*, *GetNextEditChar* returns *MaxLength*.

Use *GetNextEditChar* to loop through the editable characters in the *EditText*, starting with the value returned by *GetFirstEditChar* and using *GetLastEditChar* to determine when all nonliteral characters have been processed.

See also *EditMask* property, *EditText* property, *GetFirstEditChar* method, *GetLastEditChar* method, *GetPriorEditChar* method

GetPriorEditChar

GetPriorEditChar returns the first index of a nonliteral character equal to or less than *Offset*, where zero is the first character.

```
function GetPriorEditChar(Offset: Integer): Integer;
```

Description Call *GetPriorEditChar* to find the first position of the *EditMask* at or preceding *Offset*, where the user can modify the *EditText*. If there is no nonliteral character at or preceding *Offset*, *GetPriorEditChar* returns the index of the first editable character following *Offset*.

See also *EditMask* property, *EditText* property, *GetFirstEditChar* method, *GetLastEditChar* method, *GetNextEditChar* method

GetSel

GetSel returns the endpoints of the current selection in the *EditText*.

```
procedure GetSel(var SelStart: Integer; var SelStop: Integer);
```

Description Use *GetSel* to obtain the indexes of the first and last selected characters of the *EditText*, where 0 is the index of the first character. Not all the characters in the selection will necessarily be editable, if the mask contains literal characters. Do not use the resulting indexes on the *Text* property if there is a mask, because the *EditText* may contain blanks or nonliteral characters not found in the *Text*.

To programmatically change the selected text, use *GetNextEditChar* to cycle through the editable characters of the selection from *SelStart* to *SelStop*.

See also *EditText* property, *GetNextEditChar* method, *SelText* property, *SetSel* method, *Text* property

GetTextLen

GetTextLen returns the length of the underlying text.

```
function GetTextLen: Integer;
```

Description Use *GetTextLen* to determine the length of the underlying text for the masked edit control. The underlying text has not been formatted for display by the mask, and may differ from the *EditText* that appears in the window of the control.

See also *EditText* property, *GetTextBuf* method, *MaxLength* property, *Text* property

KeyDown

KeyDown supplements the inherited *KeyDown* to process navigation keys and the *Del* key.

```
type TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

procedure KeyDown(var Key: Word; Shift: TShiftState); override;
```

Description After calling the inherited *KeyDown*, *KeyDown* ensures that moving the cursor within the masked edit control places the cursor on an editable character, and that deleting the current selection replaces nonliteral characters with blanks. Derived objects can override this method to further supplement the processing that takes place when the user presses a key down.

See also *CheckCursor* method, *KeyPress* method, *KeyUp* method, *SetSel* method

KeyPress

KeyPress supplements the inherited method to process the *Esc* and *Enter* keys.

```
procedure KeyPress(var Key: Char); override;
```

Description After calling the inherited method, *KeyPress* calls the *Reset* method in response to the *Esc* key, or calls the *ValidateEdit* method in response to the *Enter* key. Derived classes can override this method to further supplement the processing of specific keystrokes.

See also *KeyDown* method, *Reset* method, *ValidateEdit* method

KeyUp

KeyUp supplements the inherited method to process the *Ctrl+Left* and *Ctrl+Right* key combinations.

```
TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);
procedure KeyUp(var Key: Word; Shift: TShiftState); override;
```

Description After calling the inherited method, *KeyUp* makes sure that selection using the *Ctrl+Left* and *Ctrl+Right* key combinations ends with an editable selection in the edit window. Derived classes can override this method to further supplement the processing that takes place when the user releases a key.

See also *CheckCursor* method, *KeyDown* method, *KeyPress* method, *SetSel* method

ReformatText

ReformatText applies the mask *NewMask* to the underlying text, and updates the internal representation of the mask.

```
procedure ReformatText(const NewMask: string);
```

Description *ReformatText* does the bulk of the work of setting the *EditMask* property at runtime. *ReformatText* should not be used in place of the *EditMask* property, however, as it does not deal with the *MaskState* property or the positioning of the cursor.

See also *CheckCursor* method, *EditMask* property, *EditText* property, *MaskState* property, *Text* property

Reset

Reset returns the *EditText* to its value at the time when the edit control last lost focus.

```
procedure Reset; virtual;
```

Description *Reset* cancels the most recent edits to the masked edit control. The *EditText* returns to the value at the time the control last lost focus, not the time when it first regained focus. Thus, any changes to the *EditText* that occurred programmatically while the edit control did not have focus will also be discarded.

See also *Clear* method, *EditText* property, *KeyPress* method

SetCursor

SetCursor moves the cursor to the first editable character at or before *Pos*, and selects that character.

```
procedure SetCursor(Pos: Integer);
```

Description *SetCursor* handles positioning of the cursor within the *EditText* so that it is always on an editable character. It also ensures that at least one character is always selected. Because editing a masked edit control consists of replacing the blank characters in the mask, there must always be a selection to replace. The selection corresponds to one or more nonliteral characters in the mask, which are either blank or contain characters for the user to re-enter.

See also *CheckCursor* method, *GetPriorEditChar* method, *SetSel* method

SetSel

SetSel selects characters in the *EditText* from *SelStart* to *SelStop*.

```
procedure SetSel(SelStart: Integer; SelStop: Integer);
```

Description *SetSel* selects the indicated range of characters in the *EditText*, regardless of whether any of them are editable. If *SelStart* is the same as *SelStop*, *SetSel* positions the cursor at the *SelStart* character. When using *SetSel*, leave the cursor in an editable position if the intent is to create a selection for the user to replace.

SetSel can be used to select non-editable characters that can then be captured by using the *SelText* property or by copying to the *Clipboard*. Once the selection has been captured, return the cursor to an editable position by calling *CheckCursor*.

See also *CheckCursor* method, *EditText* property, *GetSel* method, *SelText* property, *SetCursor* method

C

ValidateEdit

ValidateEdit attempts to validate the *EditText* against the current mask.

```
procedure ValidateEdit;
```

Description *ValidateEdit* attempts to validate the *EditText* after each keystroke, and when the user attempts to leave the masked edit field. If the *EditText* does not match the specifications of the *EditMask*, *ValidateEdit* calls *ValidateError* to raise an exception, and returns the cursor to the first position where the *EditText* did not comply with the *EditMask*.

See also *EditMask* property, *EditText* property, *KeyPress* method, *ValidateError* method

ValidateError

ValidateError beeps and raises an *EDBEditError* exception.

```
procedure ValidateError;
```

Description *ValidateEdit* calls *ValidateError* when the *EditText* fails to comply with the *EditMask*.

See also *EditMask* property, *EditText* property, *ValidateEdit* method

TCustomMemo

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomEdit • TCustomMemo

TCustomMemo is the abstract base type for memo components, which are multiline edit boxes, including *TMemo* and *TDBMemo*.

Description *TCustomMemo* introduces new properties and methods to expand the capabilities of *TCustomEdit* for handling multiline edit boxes.

Do not create instances of *TCustomMemo*. Use *TCustomMemo* as a base class when declaring control objects that implement a multiline edit control. Properties and methods of *TCustomMemo* provide basic behavior that descendant classes inherit as well as behavior that components can override to customize their behavior.

See also *TDBMemo* object, *TMemo* object

Properties

The following table lists the properties of *TCustomMemo*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
Alignment	protected	taLeftJustify	TCustomMemo
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Lines	public		TCustomMemo
Modified	public		TCustomEdit
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ScrollBars	protected	ssNone	TCustomMemo
SelLength	public		TCustomEdit
SelStart	public		TCustomEdit
SelText	public		TCustomEdit
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	public		TControl
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
WantReturns	protected	True	TCustomMemo
WantTabs	protected	False	TCustomMemo
Width	published		TControl
WindowProc	public		TControl
WordWrap	protected	True	TCustomMemo

Alignment

Alignment determines how the text is aligned within the edit control.

property Alignment: TAlignment;

Description Use *Alignment* to change the way the text is formatted by the edit control. *Alignment* can take one of the following values:

Value	Meaning
taLeftJustify	Align text to the left side of the control
taCenter	Center text horizontally in the control.
taRightJustify	Align text to the right side of the control.

See also *SetAlignment* method, *Text* property

Lines

Lines contains the individual lines of text in the edit control.

property Lines: TStrings;

Description Use *Lines* to manipulate text in an edit control on a line-by-line basis. *Lines* is a *TStrings* object, so the *TStrings* methods may be used for *Lines* to perform manipulations such as counting the lines of text, adding new lines, deleting lines, or replacing lines with new text.

To work with all the text at once, use the *Text* property. To manipulate individual lines of text, the *Lines* property works better.

See also *SelText* property, *SetLines* method, *Text* property, *TStrings* object, *WordWrap* property

ScrollBars

ScrollBars determines whether the edit control has any scroll bars.

property ScrollBars: TScrollStyle;

Description Use *ScrollBars* to give a multiline edit control horizontal or vertical scroll bars. *ScrollBars* can take one of the following values:

Value	Meaning
ssNone	The control has no scroll bars.
ssHorizontal	The control has a single scroll bar on the bottom edge.
ssVertical	The control has a single scroll bar on the right edge.
ssBoth	The control has a scroll bar on both the bottom and right edges.

See also *CreateParams* method, *SetScrollBars* method, *WordWrap* property

WantReturns

WantReturns determines whether the user can insert return characters into the text.

property WantReturns: Boolean;

Description Set *WantReturns* to *True* to allow users to enter return characters into the text. Set *WantReturns* to *False* to allow the form to handle return characters instead.

For example, in a form with a default button (such as an OK button) and a memo control, if *WantReturns* is *False*, pressing *Enter* chooses the default button. If *WantReturns* is *True*, pressing *Enter* inserts a return character in the text.

Note If *WantReturns* is *False*, users can still enter return characters into the text by pressing *Ctrl+Enter*.

See also *Text* property, *WantTabs* property, *WordWrap* property

WantTabs

WantTabs determines whether the user can insert tab characters into the text.

property WantTabs: Boolean;

Description Set *WantTabs* to *True* to allow users to enter tab characters into the text. Set *WantTabs* to *False* if you want the tab character to select the next control on the form instead.

Note If *WantTabs* is *True*, users can tab into the edit control, but they can't tab out.

See also *Text* property, *WantReturns* property

C

WordWrap

WordWrap determines whether the edit control inserts soft carriage returns so text wraps at the right margin.

property WordWrap: Boolean;

Description Set *WordWrap* to *True* to make the edit control wrap text at the right margin so it fits in the client area. The wrapping is cosmetic only--the text does not include any return characters that were not explicitly entered. Set *WordWrap* to *False* to have the edit control show a separate line only where return characters were explicitly entered into the text.

Note There should be no use for a horizontal scroll bar if *WordWrap* is *True*.

See also *Lines* property, *ScrollBars* property, *SetWordWrap* method, *Text* property, *WantReturns* property

Methods

The following table lists the methods of *TCustomMemo*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomEdit
ClearSelection	public	TCustomEdit
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
CopyToClipboard	public	TCustomEdit
Create	public	TCustomMemo
CreateParams	protected	TCustomMemo
CreateParented	public	TWinControl
CreateWindowHandle	protected	TCustomMemo
CutToClipboard	public	TCustomEdit
DefaultHandler	public	TObject
Destroy	public	TCustomMemo
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent

Method	Visibility	Documented in
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetSelTextBuf	public	TCustomEdit
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
KeyPress	protected	TCustomMemo
Loaded	protected	TCustomMemo
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
PasteFromClipboard	public	TCustomEdit
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectAll	public	TCustomEdit
SendToBack	public	TControl
SetAlignment	protected	TCustomMemo
SetBounds	public	TWinControl
SetFocus	public	TWinControl

Method	Visibility	Documented in
SetLines	protected	TCustomMemo
SetScrollBars	protected	TCustomMemo
SetSelTextBuf	public	TCustomEdit
SetTextBuf	public	TControl
SetWordWrap	protected	TCustomMemo
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates and initializes an instance of *TCustomMemo*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call Create to instantiate a *TCustomMemo* object. After calling the inherited constructor, *Create* adjusts some of the properties so that the control starts with the following values:

- *AutoSelect* is *True*.
- *AutoSize* is *False*.
- *BorderStyle* is *bsSingle*.
- *ControlStyle* is [*csClickEvents, csDoubleClicks, csFixedHeight, csSetCaption*]
- *Height* and *Width* are 89 and 188.
- *HideSelection* is *True*.
- *ParentColor* is *False*.
- *TabStop* is *True*.
- *WantReturns* is *True*.
- *WordWrap* is *True*.

Derived objects that override the *Create* method should call the inherited *Create* before making any additional initializations.

See also *AutoSelect* property, *AutoSize* property, *BorderStyle* property, *ControlStyle* property, *Height* property, *HideSelection* property, *ParentColor* property, *TabStop* property, *TWinControl.Create* method, *WantReturns* property, *Width* property, *WordWrap* property

CreateParams

The *CreateParams* method initializes a window-creation parameter record passed in the *Params* parameter.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
```

```
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;

procedure CreateParams(var Params: TCreateParams); override;
```

Description The *CreateWnd* method calls *CreateParams* to initialize the parameters it passes to *CreateWindowHandle*. *TCustomEdit* overrides the inherited method to specify a scrollable multiline edit control. The *Params* parameter is updated to reflect the current values of the *Alignment*, *ScrollBars*, and *WordWrap* properties, in addition to the *BorderStyle*, *HideSelections*, *PasswordChar*, and *ReadOnly* properties that the inherited method specifies.

See also *Alignment* property, *BorderStyle* property, *CreateWindowHandle* method, *CreateWnd* method, *HideSelections* property, *PasswordChar* property, *ReadOnly* property, *ScrollBars* property, *WordWrap* property

CreateWindowHandle

CreateWindowHandle creates a window handle for the edit control.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;

procedure CreateWindowHandle(const Params: TCreateParams); override
```

Description *CreateWindowHandle* overrides the inherited method to interpret the *Caption* in *Params* as the initial text rather than as the window caption.

See also *CreateParams* method, *CreateWnd* method

Destroy

Destroy destroys an instance of *TCustomMemo*

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TCustomMemo* object is not already freed, and only then calls *Destroy*.

Destroy frees the *TStrings* object that implements the *Lines* property before calling the inherited destructor.

See also *Free* method, *Lines* property, *TStrings* object

KeyPress

KeyPress overrides the default method to ensure that return characters are not entered into the text when *WantReturns* is *False*.

```
procedure KeyPress(var Key: Char); override;
```

Description After calling the inherited *KeyPress*, which calls the *OnKeyPress* event handler, *KeyPress* discards return characters when *WantReturns* is *False*.

See also *WantReturns* property

Loaded

Loaded initializes the *Modified* property to *False*.

```
procedure Loaded; override;
```

Description *Loaded* overrides the default so that the *Modified* property is set to *False* after loading the control from a stream. Derived classes can override the *Loaded* method to perform additional initializations after the control is loaded from a stream.

See also *Modified* property

SetAlignment

SetAlignment sets the *Alignment* property.

```
procedure SetAlignment(Value: TAlignment);
```

Description *SetAlignment* is exposed as a protected method so derived classes can customize their implementation of the *Alignment* property.

See also *Alignment* property

SetLines

SetLines sets the *Lines* property.

```
procedure SetLines(Value: TStrings);
```

Description *SetLines* is exposed as a protected method so derived classes can customize their implementation of the *Lines* property.

See also *Lines* property

SetScrollBars

SetScrollBars sets the *ScrollBars* property.

```
procedure SetScrollBars(Value: TScrollStyle);
```

Description *SetScrollBars* is exposed as a protected method so derived classes can customize their implementation of the *ScrollBars* property.

See also *ScrollBars* property

SetWordWrap

SetWordWrap sets the *WordWrap* property.

```
procedure SetWordWrap(Value: Boolean);
```

Description *SetWordWrap* is exposed as a protected method so derived classes can customize their implementation of the *WordWrap* property.

See also *WordWrap* property

TCustomPageProducer

Unit: HTTPApp

TObject • TPersistent • TComponent • TCustomContentProducer • TCustomPageProducer

TCustomPageProducer is the base class for objects that produce a string of HTML commands based on an input template.

Description Use *TCustomPageProducer* as a base class when defining an object that converts an HTML template into a string of HTML commands which can be interpreted by a client application such as a Web browser. HTML templates include HTML commands and HTML-transparent tags that are replaced with customized content by the page producer.

See also *TDSTableProducer* object, *TPageProducer* object

C

Properties

The following table lists the properties of *TCustomPageProducer*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Dispatcher	published		TCustomContentProducer
HTMLDoc	protected		TCustomPageProducer
HTMLFile	protected		TCustomPageProducer
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
VCLComObject	public		TComponent

HTMLDoc

HTMLDoc is the HTML template used by the page producer to generate the value returned by the *Content* method.

property HTMLDoc: TStrings;

Description Set *HTMLDoc* to a set of strings that make up the HTML template. Each string is a sequence of one or more HTML commands or HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The page producer converts the entire HTML-transparent tag into true HTML commands based on the value of the tag name. Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form ParamName=Value, where there is no space between the parameter name, the equals symbol (=) and the value. Each parameter is delimited by white space.

The angle brackets (< >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

The *Content* method converts *HTMLDoc* into a final HTML string by calling the *HandleTag* method to convert each HTML-transparent tag in *HTMLDoc*.

Note Setting *HTMLDoc* clears the *HTMLFile* property.

See also *Content* method, *HandleTag* method, *HTMLFile* property

HTMLFile

HTMLFile is the name of a file that contains the HTML template used by the page producer to generate the value returned by the *Content* method.

```
type TFileName = string;
property HTMLFile: TFileName;
```

Description Set *HTMLFile* to the name of the file that contains an HTML template. The HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The page producer converts the entire HTML-transparent tag into true HTML commands based on the value of the tag name. Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form ParamName=Value, where there is no space between the parameter name, the equals symbol (=) and the value. Each parameter is delimited by white space.

The angle brackets (< >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

The *Content* method converts the HTML template into a final HTML string by calling the *HandleTag* method to convert each HTML-transparent tag in *HTMLFile*.

Note Setting *HTMLFile* clears the *HTMLDoc* property.

Note When distributing applications that contain page producers which use the *HTMLFile* property, be sure to distribute the file along with the application. Also, be sure that the value of *HTMLFile* does not include a hard-coded path name that will not be available on the user's disk.

See also *Content* method, *HandleTag* method, *HTMLDoc* property

C

Methods

The following table lists the methods of *TCustomPageProducer*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Content	public	TCustomPageProducer

Method	Visibility	Documented in
ContentFromStream	public	TCustomPageProducer
ContentFromString	public	TCustomPageProducer
Create	public	TCustomPageProducer
DefaultHandler	public	TObject
Destroy	public	TCustomPageProducer
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HandleTag	protected	TCustomPageProducer
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Content

Content returns the HTML produced by the HTML page producer.

function Content: string; **override**;

Description Call *Content* to get the HTML commands that result from converting the HTML template. *Content* reads the template from *HTMLDoc* or *HTMLFile*, depending on which property is set. *Content* replaces each HTML-transparent tag in the HTML template with the code returned by the *HandleTag* method.

See also *HandleTag* method, *HTMLDoc* property, *HTMLFile* property

ContentFromStream

ContentFromStream returns the HTML code that results from converting an HTML template that is read from a stream.

```
function ContentFromStream(Stream: TStream): string; override;
```

Description Call *ContentFromStream* to convert an HTML template that can be read from a stream. The HTML template is a sequence of HTML commands and HTML-transparent tags. *ContentFromStream* replaces each HTML-transparent tag in the HTML template read from the stream with a string returned by the *HandleTag* method.

See also *HandleTag* method, *TStream* object

ContentFromString

ContentFromString returns the HTML code that results from converting the string *S*.

```
function ContentFromString(const S: string): string; override;
```

Description Call *ContentFromString* to convert a string that consists of an HTML template. An HTML template is a sequence of HTML commands and HTML-transparent tags. *ContentFromString* replaces each HTML-transparent tag in the HTML template with a string returned by the *HandleTag* method.

See also *HandleTag* method

Create

Create creates an instance of *TCustomPageProducer*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to create and initialize an instance of *TCustomPageProducer*. After calling the inherited constructor, *Create* allocates the *TStrings* object that implements the *HTMLDoc* property.

See also *HTMLDoc* property, *TComponent.Create* method

Destroy

Destroy destroys an instance of *TCustomPageProducer*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call Free. *Free* verifies that the page producer object is not already freed, and only then calls *Destroy*.

Destroy frees the *TStrings* object implements the *HTMLDoc* property before calling the inherited destructor.

See also *Free* method, *HTMLDoc* property

HandleTag

HandleTag converts an HTML-transparent tag into one or more HTML commands or HTML-transparent tags.

```
function HandleTag(const TagString: string; TagParams: TStrings): string; virtual;
```

Description *Applications* cannot call this protected method directly. *HandleTag* is called by the *Content* method, the *ContentFromStream* method, or the *ContentFromString* method to convert the HTML-transparent tags in an HTML template. Each HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form ParamName=Value, where there is no space between the parameter name, the equals symbol (=) and the value. Each parameter is delimited by white space.

The *TagString* parameter is the tag name portion of the HTML-transparent tag with the pound sign stripped off. The *TagParams* parameter is the parameter portion of the HTML-transparent tag. *HandleTag* returns the HTML-transparent tag with the parameters stripped off. The string returned from *HandleTag* will be ignored by Web browsers that read the resulting HTML.

See also *Content* method, *ContentFromStream* method, *ContentFromString* method

TCustomPanel

Unit: ExtCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomPanel

TCustomPanel is the abstract base class for all panel components in Delphi.

Description Use *TCustomPanel* as a base class when defining objects that include other controls as constituent parts. Panel components are controls can contain other controls, grouping them together.

When a panel is aligned with the form using the *Align* property, it maintains the same relative position to the form even when the form is resized. For example, a panel can be aligned so that it always remains on the top of the form, even when the user changes the shape and size of the form. This makes panels useful for implementing components that act like tool bars or status

bars. For example, construct a tool bar or tool palette by adding speed buttons to a panel that execute commands or set modes.

See also *TForm* object, *TPanel* object

Properties

The following table lists the properties of *TCustomPanel*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
Alignment	protected	taCenter	TCustomPanel
BevelInner	protected	bvNone	TCustomPanel
BevelOuter	protected	bvRaised	TCustomPanel
BevelWidth	protected	1	TCustomPanel
BorderStyle	protected	bsNone	TCustomPanel
BorderWidth	protected	0	TCustomPanel
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	protected	clBtnFace	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
FullRepaint	protected	True	TCustomPanel
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Locked	protected	False	TCustomPanel

Property	Visibility	Default	Documented in
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	False	TControl
ParentWindow	public		TWinControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Alignment

Alignment determines how caption is aligned within the panel.

```
type TAlignment = (taLeftJustify, taRightJustify, taCenter);
property Alignment: TAlingment;
```

Description Use the *Alignment* property to specify if the caption of a panel appears on the left, right, or center of the panel. For example, to use a panel as a status bar that displays the *TApplication.Hint* property as the panel's *Caption* property, set *Alignment* to *taLeftJustify* to place the hint text on the left side of the panel.

These are the possible values:

Value	Meaning
taLeftJustify	Align text to the left side of the panel
taCenter	Center text horizontally in the panel
taRightJustify	Align text to the right side of the panel

See also *TApplication.Hint* property, *TControl.Caption* property

BevelInner

BevelInner determines the style of the inner bevel of a panel.

```
type TPanelBevel = (bvNone, bvLowered, bvRaised);
property BevelInner: TPanelBevel;
```

Description Use *BevelInner* to provide the panel with a three-dimensional bevelled look. A panel component has two bevels, an outer bevel drawn next to the border of the control, and an inner bevel drawn inside the outer bevel, separated by the number of pixels specified in the *BorderWidth* property. The width of the bevels, if they are drawn, is the *BevelWidth* property.

The *BevelInner* property determines the style of the inner bevel of a panel component. These are the possible values:

Value	Meaning
bvNone	No inner bevel exists.
bvLowered	The inner bevel is lowered.
bvRaised	The inner bevel is raised.

See also *BevelOuter* property, *BevelWidth* property, *BorderWidth* property

BevelOuter

BevelOuter determines the style of the outer bevel of a panel.

```
type TPanelBevel = (bvNone, bvLowered, bvRaised);
property BevelOuter: TPanelBevel;
```

Description Use *BevelOuter* to provide the panel with a three-dimensional bevelled look. A panel component has two bevels, an outer bevel drawn next to the border of the control, and an inner bevel drawn inside the outer bevel, separated by the number of pixels specified in the *BorderWidth* property. The width of the outer bevel is specified in the *BevelWidth* property in pixels.

These are the possible values:

Value	Meaning
bvNone	No outer bevel exists.
bvLowered	The outer bevel is lowered.
bvRaised	The outer bevel is raised.

See also *BevelInner* property, *BevelWidth* property, *BorderWidth* property

BevelWidth

BevelWidth determines the width, in pixels, of both the inner and outer bevels of a panel.

```
type TBevelWidth = 1..MaxInt;
property BevelWidth: TBevelWidth;
```

Description Use *BevelWidth* to specify how wide the inner or outer bevel should be. Don't confuse *BevelWidth*, which is the width of the bevels, with *BorderWidth*, which is the space between the bevels.

If both the *BevelInner* and *BevelOuter* properties are *bvNone*, *BevelWidth* has no effect. To remove both bevels, set the *BevelInner* and *BevelOuter* properties to *bvNone*, rather than setting the *BevelWidth* to 0, as this involves less overhead when painting.

See also *BevelInner* property, *BevelOuter* property, *BorderWidth* property

BorderStyle

BorderStyle determines the style of the line drawn around the perimeter of the panel control.

```
property BorderStyle: TBorderStyle;
```

Description Use *BorderStyle* to specify whether the panel has a single line drawn around it. These are the possible values:

Value	Meaning
bsNone	No visible border
bsSingle	Single-line border

Don't confuse the line drawn around the panel with the *BorderWidth* of the panel. The *BorderWidth* of the panel is the distance between the outer and inner bevels.

See also *BorderWidth* property

BorderWidth

BorderWidth is the distance, in pixels, between the outer and inner bevels.

```
type TBorderWidth: 0..MaxInt;

property BorderWidth: TBorderWidth;
```

Description Use *BorderWidth* to specify how wide the border around the panel should be. A value of 0 means no border should appear.

The border of a panel is the area between the outer and inner bevels. It is visible only if the inner bevel is raised or lowered, but affects the inset of the caption within the panel even if *BevelInner* is *bvNone*. If the *Alignment* property is not *taCenter*, the *Caption* will be aligned to the inner edge of the border. This edge is *BorderWidth* pixels in from the outer bevel if *BevelInner* is *bvNone*. It is the inner edge of the inner bevel otherwise.

Don't confuse the border of the panel with line drawn around the panel itself. The line around the panel is specified by the *BorderStyle* property.

See also *Alignment* property, *BevelInner* property, *BevelOuter* property, *BorderStyle* property, *TControl.Caption* property

FullRepaint

FullRepaint determines whether the *Invalidate* method is called when the custom panel receives a WM_SIZE message.

```
property FullRepaint: Boolean;
```

Description Set *FullRepaint* to *True* to cause the panel and all the controls it contains to be repainted when the panel is resized. If *FullRepaint* is *False*, the panel and its controls are not repainted.

See also *Invalidate* method, *OnResize event*, *Resize* method

Locked

Locked determines whether a panel that is being used as a toolbar is replaced by a toolbar supplied by an OLE server.

```
property Locked: Boolean;
```

Description Set the *Locked* property to *False* when the panel is used as a toolbar by a *TOleContainer* component. Any panels aligned to a *TOleContainer* are assumed to be replaceable toolbars unless the *Locked* property is *True*. Prevent the panel from being replaced by an OLE server's toolbar by setting the panel's *Locked* property to *True*.

Note An OLE application's toolbars must be in panels aligned to a side of the form (for example, their *Align* properties must be set to *alTop*, *alBottom*, *alLeft*, or *alRight*).

See also *Align* property, *TOleContainer* object

C

Events

The following table lists the events of *TCustomPanel*:

Event	Visibility	Documented in
OnResize	protected	TCustomPanel

OnResize

OnResize occurs when the panel is resized while an application is running.

```
TNotifyEvent = procedure (Sender: TObject) of object;

property OnResize: TNotifyEvent;
```

Description Write an *OnResize* event handler to make something happen when the panel is resized. For example, the panel may want to reposition its child controls when it is resized.

See also *AlignControls* method, *Resize* method

Methods

The following table lists the methods of *TCustomPanel*:

Method	Visibility	Documented in
AlignControls	protected	TCustomPanel
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomPanel
CreateParams	protected	TCustomPanel
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCustomControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject

Method	Visibility	Documented in
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Paint	protected	TCustomPanel
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
Resize	protected	TCustomPanel
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl

Method	Visibility	Documented in
Update	public	TWinControl
UpdateControlState	public	TWinControl

AlignControls

AlignControls aligns any controls contained within the panel.

procedure AlignControls(AControl: TControl; **var** Rect: TRect); **override**;

Description Use *AlignControls* to align all controls within a panel all at once. Specify the area in which you want the controls aligned as the value of the *Rect* parameter.

AlignControls uses the *Align* property of each child control to determine how to align it. The control passed as the *AControl* parameter takes precedence in alignment over other, similarly-aligned controls. The *AControl* parameter can be **nil**.

See also *TControl.Align* property

Create

Create creates and initializes an instance of *TCustomPanel*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a panel at runtime. Panels placed in forms at design time are created automatically.

After calling the inherited constructor, Create initializes the following properties:

- *Alignment* to *taCenter*.
- *BevelOuter* to *bvRaised* and *BevelWidth* to 1.
- *BorderStyle* to *bsNone*.
- *Color* to *BtnFace*.
- *ControlStyle* to [*csAcceptsControls, csCaptureMouse, csClickEvents, csOpaque, csSetCaption, csDoubleClick, csReplicatable*].
- *FullRepaint* to *True*.
- *Height* to 41 and *Width* to 185.

See also *Alignment* property, *BevelOuter* property, *BevelWidth* property, *BorderStyle* property, *ControlStyle* property, *FullRepaint* property, *Height* property, *TControl.Color* property, *TCustomControl.Create* method, *Width* property

CreateParams

CreateParams initializes a window-creation parameter record passed in the *Params* parameter.

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description *CreateParams* is called automatically to initialize the window-creation parameters whenever the window for the panel needs to be created. *CreateParams* overrides the inherited method to implement the *BorderStyle* property.

See also *BorderStyle* property, *TWinControl.CreateParams* method

Paint

Paint renders the image of the panel.

```
procedure Paint; override;
```

Description Override the *Paint* method to change the way the panel is drawn. The *Paint* method for custom panels draws the bevels of the panel, sets the brush and font for the panel, and fills in the interior of the panel before calling the Windows API *DrawText* function to draw the *Caption*.

See also *BevelInner* property, *BevelOuter* property, *Brush* property, *TControl.Caption* property, *TCustomControl.Canvas* property

Resize

Resize is the protected implementation method for the *OnResize* event.

```
procedure Resize; dynamic;
```

Description Custom controls call *Resize* in response to the WM_SIZE message, indicating a change in the size of the control or form. Override *Resize* to provide additional responses in addition to the call to the *OnResize* event handler.

If the panel's *FullRepaint* property is *True*, *Resize* invalidates the panel before it calls the method assigned to the *OnResize* event, if one exists.

See also *FullRepaint* property, *Invalidate* method, *OnResize* event

TCustomProvider

Unit: Provider

TObject • TPersistent • TComponent • TCustomProvider

Provides an abstract, virtualized base class for deriving customized data provision objects for an application server in a multi-tiered database application.

Description

Use *TCustomProvider* to derive customized data provider objects for application servers in a multi-tiered database application that bypasses the Borland Database Engine (BDE). *TCustomProvider* defines the fundamental behaviors required of all provider objects, either through abstract method declarations, or through virtual declarations and implementations that are overridden in subsequently derived objects.

In general, a provider derived from *TCustomProvider* is an intermediate link between a *TClientDataSet* object on the client, and a dataset object on the application server. It packages data from the server dataset and passes it in one or more transportable blocks to the client dataset.

The client dataset receives the blocked data, and reconstructs it to create a local, in-memory copy of the data for user access. When user access is complete, the client dataset repackages any changed data and sends the updates to the provider on the application server. The provider applies the updates to the remote database server.

TProvider is an example of a BDE-enabled provider derived from *TCustomProvider*.

See also

TClientDataSet object, *TProvider* object

Properties

The following table lists the properties of *TCustomProvider*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Constraints	protected		TCustomProvider
Data	protected, read-only		TCustomProvider
DesignInfo	public		TComponent
Name	published		TComponent

Property	Visibility	Default	Documented in
Owner	public, read-only		TComponent
Provider	protected, read-only		TCustomProvider
Tag	published	0	TComponent
VCLComObject	public		TComponent

Constraints

Controls whether or not database constraints are applied to client requests for data.

property Constraints: WordBool;

Description Set *Constraints* to specify whether or not constraints defined for the database table or tables underlying a database application server's dataset components are applied to client requests for data. When *Constraints* is *True* (the default), constraints are applied. To disable constraints, set *Constraints* to *False*.

Data

Returns a data packet created by the provider.

property Data: OleVariant;

Description *Data* is a read-only property that is used to create a data packet containing all records. As implemented for *TCustomProvider*, *Data* calls *GetRecords* with a parameter of *–1*. *GetRecords* returns a null.

Descendants of *TCustomProvider* must reimplement the methods that get and set *Data*.

See also *Get_Data* method

Provider

Points to the provider interface implemented by the provider component.

property Provider: IProvider;

Description *Provider* is a read-only property that points to the provider interface implemented by the provider component to communicate with a client dataset.

Events

The following table lists the events of *TCustomProvider*:

Event	Visibility	Documented in
OnDataRequest	protected	TCustomProvider

OnDataRequest

OnDataRequest occurs when a provider calls its *DataRequest* method.

Declaration

```
type
  TDataRequestEvent = function(Sender: TObject; Input : OleVariant) : OleVariant of object;

property OnDataRequest: TDataRequestEvent;
```

Write an *OnDataRequest* event handler to do custom processing of data. An application need not use this event handler to work with provider data.

If an application does not provide an *OnDataRequest* event handler, calls to *DataRequest* return null.

See also *DataRequest* method

Methods

The following table lists the methods of *TCustomProvider*:

Method	Visibility	Documented in
ApplyUpdates	protected	TCustomProvider
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TComponent
DataRequest	protected	TCustomProvider
DefaultHandler	public	TObject
Destroy	protected	TCustomProvider
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent

Method	Visibility	Documented in
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetMetaData	protected	TCustomProvider
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetProvider	protected	TCustomProvider
GetRecords	protected	TCustomProvider
Get_Data	protected	TCustomProvider
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
Reset	protected	TCustomProvider
SafeCallException	public	TComponent

ApplyUpdates

Declares an **abstract, virtual** method that applies updates received from a client dataset to datasets connected to a remote database server.

```
function ApplyUpdates(Delta: OleVarant; MaxErrors: Integer; out ErrorCount: Integer);
    OleVariant; virtual; abstract;
```

Description

ApplyUpdates is not implemented for *TCustomProvider*; it is only declared.

Implement *ApplyUpdates* in a derived provider component to enable an application server to write a client's updates to a remote database server. *Delta* is a variant containing the updated, inserted, and deleted records to write to the database. *MaxErrors* indicates the maximum number of errors to permit before stopping the update operation; *ErrorCount* indicates the actual number of errors encountered during the update operation.

ApplyUpdates should return a variant that is a data packet containing all records that could not be applied to the database.

DataRequest

Calls the *OnDataRequest* event handler of the provider.

```
function DataRequest(Input: OleVariant): OleVariant;
```

Description Write an *OnDataRequest* event handler and call the *DataRequest* function to invoke the event handler at runtime. This function and the event it invokes enables an application to perform custom processing in response to a request for data.

Note Application code must explicitly call *DataRequest*. Delphi does not call *DataRequest* itself.

See also *OnDataRequest* event

Destroy

Frees an instance of a provider object.

```
destructor Destroy;
```

Description Do not call *Destroy* directly. Instead, call *Free*, which checks that an object is not already freed before calling its *Destroy* method. *Destroy* frees all *IProvider* objects associated with the *TCustomProvider* object and calls its inherited *Destroy* method.

See also *Create* method

Get_Data

Retrieves records from the provider's dataset source in response to a client request.

```
function Get_Data: OleVariant; virtual;
```

Description Call *Get_Data* to retrieve data in response to a client request. *Get_Data* retrieves all records at once from the provider's dataset. The provider retrieves data from a dataset source, assembles it into a data packet, and passed the packet to the client as a variant.

Get_Data calls *GetRecords* with a parameter of *–1*.

See also *GetMetaData* method, *GetRecords* method

GetMetaData

Retrieves a data packet containing a dataset's metadata.

```
function GetMetaData: variant;
```

Description Call *GetMetaData* to retrieve a dataset's metadata. *GetMetaData* calls *GetRecords* with a parameter of *0*.

See also *Get_Data* method, *GetRecords* method

GetProvider

Creates a new provider interface object for the provider.

```
function GetProvider: IProvider;
```

Description Call *GetProvider* to establish a new provider interface (*IProvider*) for a provider component. Each call to *GetProvider* creates a new provider interface and adds that interface to an internal list of *IProvider* interfaces maintained by the provider component.

GetRecords

Retrieves data from a provider's dataset.

```
function GetRecords(Count: Integer; out RecsOut: Integer); OleVariant; virtual;
```

Description Override *GetRecords* in derived provider components to enable an application to retrieve a specified number of records from the provider's dataset. *Count* indicates the number or type of records to retrieve. If *Count* is –1, all records should be retrieved. If *Count* is *0*, only metadata should be retrieved. If *Count* is greater than *0*, only the specified number of records should be retrieved. *RecsOut* indicate the actual number of records retrieved. Records are returned as a data packet in a variant.

A client application calls a provider's *GetRecords* method to retrieve a partial set of all available records from a dataset. *GetRecords* is especially useful when dealing with very large datasets, or when working with datasets that have blob fields.

As implemented for *TCustomProvider*, *GetRecords* always returns NULL.

See also *DataRequest* method, *Get_Data* method, *GetMetaData* method

Reset

Implements a virtual method for restarting record fetching at the start of a dataset.

```
procedure Reset; virtual;
```

Description Override *Reset* in derived provider components to enable an application to start fetching records from the beginning of a dataset after having fetched sequential blocks of records with repeated calls to *GetRecords*. Otherwise each fetch operation starts at the record following the last record fetched in the previous call to *GetRecords*.

As implemented for *TCustomProvider*, *Reset* does nothing.

See also *GetRecords* method

TCustomRadioGroup

Unit: ExtCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomGroupBox • TCustomRadioGroup

TCustomRadioGroup is the base class for radio-group components.

Description Radio groups are collections of radio buttons in a single group box. *TCustomRadioGroup* is the class from which radio-group components—including *TRadioGroup* and *TDBRadioGroup*—descend.

Radio buttons that are contained directly in the same control component are said to be "grouped." When the user checks a radio button, all other radio buttons in its group become unchecked. Hence, two radio buttons on a form can be checked at the same time only if they are placed in separate containers, such as group boxes.

See also *TRadioGroup* object, *TDBRadioGroup* object, *TRadioButton* object

Properties

The following table lists the properties of *TCustomRadioGroup*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Columns	protected	1	TCustomRadioGroup
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl

Property	Visibility	Default	Documented in
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ItemIndex	protected	–1	TCustomRadioGroup
Items	protected		TCustomRadioGroup
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Columns

The number of columns in the radio group.

property Columns: Integer

Description The *Columns* property determines the number of columns in the radio group. Its value can range from 1 to 16. The default value is 1, which means that the radio buttons are arranged in a single vertical line.

ItemIndex

Indicates which radio button in the group is currently selected.

property ItemIndex: Integer

Description *ItemIndex* holds the ordinal number of the selected radio button in the *Items* list. (The first button is 0.) The value of *ItemIndex* changes at runtime as the user selects radio buttons. If you want one of the buttons to appear selected when the application starts, assign that button to *ItemIndex* at design time; otherwise, leave *ItemIndex* set to the default value of –1, which means that no button is selected.

Items

A list of the radio buttons in the radio group.

property Items: TStrings

Description *Items* holds a *TStrings* object that lists the captions of the radio buttons in the group. In *TCustomRadioGroup* descendants *TRadioGroup* and *TDBRadioGroup*, where this property is published, radio buttons can be added or removed by editing the *Items* list from the Object Inspector.

Note The items in a radio-group object are special instances of *TRadioButton* generated by *TCustomRadioGroup*. Independently created *TRadioButton* instances cannot be included in a radio group.

See also *TStrings* object, *TRadioGroup* object, *TDBRadioGroup* object, *TRadioButton* object

Methods

The following table lists the methods of *TCustomRadioGroup*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
CanModify	protected	TCustomRadioGroup
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomRadioGroup
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TCustomRadioGroup
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject

Method	Visibility	Documented in
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetChildren	protected	TCustomRadioGroup
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
ReadState	protected	TCustomRadioGroup
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent

Method	Visibility	Documented in
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

CanModify

Determines whether the user can select a different radio button in the group.

function CanModify: Boolean; **virtual**;

Description Event handlers call *CanModify* to verify that the user is allowed to select a different radio button in the group. As defined in *TCustomRadioGroup*, this method always returns *True*; override it to return *False* in situations where the user should not be able to make changes.

See also *TDBRadioGroup.CanModify* method

Create

Creates and initializes a *TCustomRadioGroup* instance.

constructor Create(AOwner: TComponent); **override**;

Description *Create* calls the inherited constructor method and initializes *TCustomRadioGroup*'s properties. It sets the *ControlStyle* property's *csSetCaption* and *csDoubleClicks* flags.

See also *TControl* object, *ControlStyle* property, *TControlStyle* type

Destroy

Destroys the *TCustomRadioGroup* instance and the radio buttons it contains.

destructor Destroy; **override**;

Description *Destroy* frees the memory for each item in the group, then calls the inherited destructor method.

GetChildren

The *GetChildren* method for *TCustomRadioGroup* is inoperative.

procedure GetChildren(Proc: TGetChildProc; Root: TComponent); **override**;

Description *TCustomRadioGroup* overrides *TWinControl* to disable the *GetChildren* method.

See also *TWinControl.GetChildren* method, *TComponent.GetChildren* method, *TComponent.GetParentComponent* method

ReadState

Reads the radio group's data from a stream.

procedure ReadState(Reader: TReader); **override**;

Description *ReadState* calls the inherited *ReadState* method to read the radio group's published properties and other stored data to a reader object. It then redraws the buttons in the group.

See also *TComponent.ReadState* method, *TControl.ReadState* method, *TWinControl.ReadState* method

TCustomRemoteServer

Unit: DBClient

TObject • TPersistent • TComponent • TCustomRemoteServer

TCustomRemoteServer is the base class for a remote server connection in a client application that is part of a multi-tiered application.

Description *TCustomRemoteServer* is the base class object from which a developer can derive a remote server connection object for a client application that is part of a multi-tiered database application. Remote server objects derived from *TCustomRemoteServer* establish a connection between a client application and an application server.

Developers should not instantiate *TCustomRemoteServer* objects in their applications. Instead they should either use the *TRemoteServer* object to establish connections, or they should derive their own custom connection object from *TCustomRemoteServer*.

See also *TClientDataSet* object, *TProvider* object, *TRemoteServer* object

Properties

The following table lists the properties of *TCustomRemoteServer*:

Property	Visibility	Default	Documented in
ClientDataSets	protected, read-only		TCustomRemoteServer
ComObject	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Connected	protected	False	TCustomRemoteServer
DesignInfo	public		TComponent
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
VCLComObject	public		TComponent

ClientDataSets

Lists client datasets associated with the remote server.

```
property ClientDataSets: TList;
```

Description Use *ClientDataSets* to examine the list of *TClientDataSet* components that use this remote server component's connection to establish links to provider objects.

See also *AddDataSet* method

Connected

Connected indicates whether or not the remote server object is connected to a remote application server.

```
property Connected: Boolean;
```

Description Use *Connected* to read or set the current connection status for a remote server object. By default, *Connected* is *False* because the *GetConnected* method that reads the current value of *Connected* always returns *False*.

Developers who derive their own remote server objects from *TCustomRemoteServer* should override *GetConnected* to return *True* when a connection is established, and should override *DoConnect and DoDisconnect*.

See also *DoConnect* method, *DoDisConnect* method, *GetConnected* method, *SetConnected* method

Events

The following table lists the events of *TCustomRemoteServer*:

Event	Visibility	Documented in
AfterConnect	protected	TCustomRemoteServer
AfterDisconnect	protected	TCustomRemoteServer
BeforeConnect	protected	TCustomRemoteServer
BeforeDisconnect	protected	TCustomRemoteServer

AfterConnect

AfterConnect occurs after a remote server establishes a connection with an application server.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property AfterConnect: TNotifyEvent;
```

Description Write an *AfterConnect* event handler to take application-specific actions after a remote server establishes a connection to an application server.

See also *BeforeConnect* event

AfterDisconnect

AfterDisconnect occurs after a remote server drops a connection to an application server.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property AfterDisconnect: TNotifyEvent;
```

Description Write an *AfterDisconnect* event handler to take application-specific actions after a remote server drops a connection to an application server.

See also *BeforeDisconnect* event

BeforeConnect

BeforeConnect occurs before a remote server establishes a connection with an application server.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property BeforeConnect: TNotifyEvent;
```

Description Write an *BeforeConnect* event handler to take application-specific actions before a remote server establishes a connection to an application server.

See also *AfterConnect* event

BeforeDisconnect

BeforeDisconnect occurs before a remote server drops a connection to an application server.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property BeforeDisconnect: TNotifyEvent;
```

Description Write an *BeforeDisconnect* event handler to take application-specific actions before a remote server drops a connection to an application server.

See also *BeforeDisconnect* event

Methods

The following table lists the methods of *TCustomRemoteServer*:

Method	Visibility	Documented in
AddDataSet	protected	TCustomRemoteServer
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TCustomRemoteServer
DefaultHandler	public	TObject
Destroy	public	TCustomRemoteServer
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
DoConnect	protected	TCustomRemoteServer
DoDisconnect	protected	TCustomRemoteServer
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent

Method	Visibility	Documented in
GetConnected	protected	TCustomRemoteServer
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetProvider	protected	TCustomRemoteServer
GetProviderNames	protected	TCustomRemoteServer
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
Loaded	protected	TCustomRemoteServer
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
RemoveDataSet	protected	TCustomRemoteServer
SafeCallException	public	TComponent
SetConnected	protected	TCustomRemoteServer

AddDataSet

Adds a client dataset to those known by the server connection.

```
procedure AddDataSet(DataSet: TClientDataSet);
```

Description *AddDataSet* adds a specified client dataset to those that can use this remote server's connection to an application server. This procedure is called automatically by client dataset objects.

See also *RemoveDataSet* method, *TClientDataSet* object

Create

Instantiates an instance of a remote server component.

```
constructor Create(AOwner: TComponent);
```

Description *Create* instantiates an instance of a remote server component. It calls its inherited constructor and creates an empty list for storing the names of client datasets that use this instance of the remote server.

See also *Destroy* method

Destroy

Frees this instance of the remote server component.

```
destructor Destroy;
```

Description Do not call *Destroy* directly. Instead, call *Free*, which determines that the component is not already freed before calling *Destroy*. *Destroy* drops the connection to an application server, frees the component's list of client datasets, and calls its inherited destructor.

See also *Create* method

DoConnect

DoConnect is a **virtual** procedure for establishing a connection to an application server.

```
procedure DoConnect; virtual
```

Description Developers who derive custom remote server components should implement the *DoConnect* method to establish a connection from the remote server component to an application server that brokers data through provider interfaces. As implemented for *TCustomRemoteServer*, *DoConnect* is an empty procedure.

For an example of a *DoConnect* implementation using ActiveX, and a dual interface, see the source code for *TRemoteServer*.

See also *AfterConnect* event, *BeforeConnect* event, *TRemoteServer.DoConnect* method

DoDisconnect

DoDisconnect is a **virtual** procedure for dropping a connection to an application server.

```
procedure DoDisconnect; virtual
```

Description Developers who derive custom remote server components should implement the *DoDisconnect* method to drop a connection between the remote server component and an application server that brokers data through provider interfaces. As implemented for *TCustomRemoteServer*, *DoDisconnect* is an empty procedure.

See also *AfterDisconnect* event, *BeforeDisconnect* event

GetConnected

GetConnected is a **virtual** function intended to serve as the read method for the *Connected* property.

```
function GetConnected: Boolean;
```

Description Developers who derive custom remote server objects from *TCustomRemoteServer* should override this function to read the current value of the *Connected* property. As defined for *TCustomRemoteServer*, *GetConnected* always returns *False*.

See also *Connected* property, *SetConnected* method

GetProvider

GetProvider is an **abstract, virtual** function that should return a pointer to the *IProvider* interface established by the provider object on the application server that is subsequently used by a client dataset to communicate directly with a provider object for data exchange.

```
function GetProvider (const ProviderName: string): IProvider; virtual;
```

Description Developers who derive custom remote server objects from *TCustomRemoteServer* should implement this function to return the *IProvider* interface established by the application server when a client application connects to the server. The interface is used by client datasets to request data from the server and send updates to the server to be applied to a remote database server.

See also *Connected* property, *TClientDataSet* object, *TProvider* object

GetProviderNames

GetProviderNames is a **virtual** procedure that is intended to retrieve a list of all available provider objects from an application server into a list created and maintained by the remote server.

```
procedure GetProviderNames(List: TStrings); virtual;
```

C

Description Developers who derive custom remote server objects from *TCustomRemoteServer* should implement this procedure to return a list of all available *IProvider* interfaces established by the application server when this client application connects to the server. Interfaces can then be made available to client datasets that use this remote server.

As implemented for *TCustomRemoteServer*, *GetProviderNames* is an empty procedure.

See also *GetProvider* method

Loaded

Loaded prepares a remote server component for connecting to an application server during form loading.

```
procedure Loaded;
```

Description Applications should have no need to call *Loaded*.This procedure is called automatically by Delphi. *Loaded* prepares a remote server component for connecting to an application server when an application is loaded.

See also *SetConnected* method

RemoveDataSet

Removes a client dataset from those known by the server connection.

```
procedure RemoveDataSet(DataSet: TClientDataSet);
```

Description *RemoveDataSet* removes a specified client dataset from its list of those that can use this remote server's connection to an application server. This procedure is called automatically by client dataset objects.

See also *AddDataSet* method, *TClientDataSet* object

SetConnected

SetConnected establishes or drops a connection between a client application and a remote application server.

```
procedure SetConnected (Value: Boolean);
```

Description Call *SetConnected* to establish or drop a connection from the remote server to an application server. If *Value* is *True*, *SetConnected* calls the *BeforeConnect* event handler, establishes a connection, and calls the *AfterConnect* event handler. If *Value* is *False*, *SetConnected* calls the *BeforeDisconnect* event handler, drops the connection, and calls the *AfterDisconnect* event handler.

See also *Connected* property

TCustomRichEdit

Unit: ComCtrls

TObject • TPersistent • TComponent • TControl•TWinControl • TCustomEdit • TCustomMemo • TCustomRichEdit

The *TCustomRichEdit* object is the abstract base type for rich edit controls such as *TRichEdit*.

Description *TCustomRichEdit* introduces new properties and methods to expand the capabilities of *TCustomMemo* for rich text.

Do not create instances of *TCustomRichEdit*. Use *TCustomRichEdit* as a base class when declaring control objects that implement rich text capabilities in an edit control. Properties and methods of *TCustomRichEdit* provide basic behavior that descendent classes inherit as well as behavior that components can override to customize their behavior.

See also *TConversion* object, *TParaAttributes* object, *TRichEdit* object, *TTextAttributes* object

Properties

The following table lists the properties of *TCustomRichEdit*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DefAttributes	public		TCustomRichEdit
DefaultConverter	public		TCustomRichEdit
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
HideScrollBars	protected	True	TCustomRichEdit
HideSelection	protected	True	TCustomRichEdit
Hint	published		TControl
Left	published		TControl
Lines	protected		TCustomRichEdit
Modified	public		TCustomEdit
Name	published		TControl
Owner	public, read-only		TComponent
PageRect	public		TCustomRichEdit
Paragraph	public, read-only		TCustomRichEdit

C

Property	Visibility	Default	Documented in
Parent	public		TControl
ParentWindow	public		TWinControl
PlainText	protected	False	TCustomRichEdit
SelAttributes	public		TCustomRichEdit
SelLength	public		TCustomRichEdit
SelStart	public		TCustomRichEdit
SelText	public		TCustomRichEdit
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	-1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	public		TControl
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

DefAttributes

DefAttributes is a *TTextAttributes* object that describes the rich text characteristics of the default font for the rich edit control

```
property DefAttributes: TTextAttributes;
```

Description

Use *DefAttributes* to discover or set the default font characteristics that the rich edit control uses for newly inserted text. These are the characteristics of the text before any special attributes have been applied. Once any special attributes are applied to a portion of text, no text from that point on is considered to have the default attributes, even if the attributes match the *DefAttributes*.

For example, if a rich edit control has had no attributes applied, the entire text has the default attributes. Selecting a portion of text in the middle and applying a specific attribute (such as a font change) results in three sections: a first section with the default attributes, a middle section with the applied attribute, and a final section with a non-default set of attributes that match the default attributes. Changing the *DefAttributes* property affects only the first section.

When inserting new text, the font characteristics of the new text will match the font characteristics at the cursor position, or, if the typing replaces a selection, the font characteristics of the selected text. New text will only have the default attributes when typed into the section of text that has the default attributes.

Note *DefAttributes* is available only at runtime.

See also *PlainText* property, *SelAttributes* property, *TTextAttributes* object

DefaultConverter

DefaultConverter is the actual class of the *TConversion* object that is used to convert between native file format and the internal format of a rich text control, when working with files that do not have a registered extension.

```
type TConversionClass = class of TConversion;

property DefaultConverter: TConversionClass;
```

Description Use *DefaultConverter* to specify the type of conversion to use when streaming text to or from files that do not have a registered extension. If *DefaultConverter* is not set, *TCustomRichEdit* uses *TConversion*, which performs no actual conversion. *TCustomRichEdit* encodes text attributes in rich-text format (RTF). Thus, unrecognized file extensions lead to no conversion, which works for files that contain plain text or any subset of RTF format.

To support streaming to or from files that encode rich text characteristics in some other format, register a conversion format with the file extension used by those files. Use *DefaultConverter* only to change the conversion that is used for unrecognized file extensions.

Note Use the *Lines* property to stream the text to or from a file.

See also *Lines* property, *PlainText* property, *RegisterConversionFormat* method, *TConversion* object

HideScrollBars

HideScrollBars determines whether the scroll bars disappear when not needed.

```
property HideScrollBars;
```

Description Use *HideScrollBars* to determine whether the scroll bars disappear from the edit control when the text appears completely within the edit window. Set *HideScrollBars* to *False* to prevent the scroll bars from flashing on and off when the contents of the rich edit control change.

HideScrollBars does nothing when the *ScrollBars* property is set to *ssNone* (The default for *TRichEdit*).

See also *ScrollBars* property

HideSelection

HideSelection determines whether the visual indication of the selected text remains when focus shifts to another control.

property HideSelection: Boolean;

Description Set *HideSelection* to *False* when it is important to provide visual feedback of the selected portion of the text even when the edit control does not have focus. Set *HideSelection* to *True* to make the visual indication of selection appear only when the edit control has focus. *HideSelection* does not affect the actual value of the selection, only the visual indication. Always setting *HideSelection* to *False* can make forms containing many edit controls look busy.

See also *SelLength* property, *SelStart* property, *SelText* property, *SetSelTextBuf* method

Lines

Lines contains the individual lines of text in the rich text edit control.

property Lines: TStrings;

Description Use *Lines* to manipulate the text in the rich text edit control on a line by line basis. *Lines* is a *TStrings* object, so *TStrings* methods may be used for *Lines* to perform manipulations such as counting the lines of text, adding lines, deleting lines, or replacing the text in lines.

To work with the text as one chunk, use the *Text* property. To manipulate individual lines of text, the *Lines* property works better.

See also *SelText* property, *Text* property, *TStrings* object, *WordWrap* property

PageRect

PageRect specifies the dimensions, in twips, of the logical page size used when printing the contents of a rich text edit control.

property PageRect: TRect;

Description Use *PageRect* to specify the logical page size used when formatting the contents of the rich edit control for printing. Specify the dimensions in twips (1/20 of a point).

See also *Print* method

Paragraph

Paragraph specifies the formatting information for the current paragraphs.

property Paragraph: TParaAttributes;

Description Read *Paragraph* to get the *TParaAttributes* object used by the rich edit control to specify paragraph formatting information. Use the *TParaAttributes* object to read or write the paragraph formatting information for the current paragraph. Paragraph formatting information includes alignment, indentation, numbering, and tabs.

Paragraph is a read-only property, because a *TCustomRichEdit* object has only one *TParaAttributes* object, which cannot be changed. The attributes of the current paragraphs, however, can be changed, by setting the properties of the *TParaAttributes* object. These can be set one by one, or all can be set to the value of an existing *TParaAttributes* object by using *Paragraph.Assign*.

The current paragraphs are the paragraphs that contain the selected text. If no text is selected, the current paragraph is the one containing the cursor.

Note *Paragraph* is available only at runtime.

See also *OnSelectionChange* event, *SelAttributes* property, *SelStart* property, *SelText* property, *TParaAttributes* object

PlainText

PlainText controls whether the rich edit control treats the text as plain text or rich text when streaming to or from a file.

```
property PlainText: Boolean;
```

Description To write the rich text in the control to a plain text file, set *PlainText* to *True* before streaming the text to a file. To ignore the rich text information encoded in a file, set *PlainText* to *True* before streaming the text to the control. To stream in the rich text attributes encoded in a file, or save the encoding of the rich text attributes to a file, set *PlainText* to *False*.

If the rich text attributes of a file are encoded in some format other than rich text format (RTF), it is necessary to use a converter on the text, even when *PlainText* is *True*.

Note Rich edit controls do not directly support streaming. Use the *Lines* property to stream to or from a file.

See also *DefaultConverter* property, *Lines* property, *RegisterConversionFormat* method

C

SelAttributes

SelAttributes is a *TTextAttributes* object that describes the rich text characteristics of the selected text in the rich edit control.

```
property SelAttributes: TTextAttributes;
```

Description Use *SelAttributes* to discover or set the font characteristics of the currently selected text. *SelAttributes* is a *TTextAttributes* object, which specifies characteristics such as font face, color, size, style, and pitch. To change a single attribute of the currently selected text, read *SelAttributes*, and set one of its properties. To change all of the attributes of the currently selected text, set *SelAttributes* to a *TTextAttributes* object that represents the desired configuration of attributes. If no text is selected, *SelAttributes* represents the attributes of the cursor position.

When inserting new text, the font characteristics of the new text will match *SelAttributes*.

Note *SelAttributes* is available only at runtime.

See also *DefAttributes* property, *OnProtectChange* event, *PlainText* property, *TTextAttributes* object

SelLength

SelLength is the number of characters that are selected.

```
property SelLength: Integer;
```

Description Read *SelLength* to determine the length, in characters, of the selected text. Set *SelLength* to change the selection to consist of the first *SelLength* characters starting at *SelStart*.

Note Setting *SelLength* to a value greater than the number of characters from *SelStart* to the end of the text results in the selection of all characters from *SelStart* to the end of the text. Reading *SelLength* immediately after setting it to a value greater than the number of available characters returns the number of characters actually selected, not the value that was just set.

See also *SelStart* property, *SelText* property

SelStart

SelStart is the position of the first selected character in the text.

```
property SelStart: Integer;
```

Description Read *SelStart* to determine the position of the first selected character, where 0 indicates the first character. If there is no selected text, *SelStart* indicates the position of the cursor. Set *SelStart* to remove the current selection and position the cursor just before the indicated character.

To select a particular range of the text, first set *SelStart* to position the cursor, and then set *SelLength* to extend the selection.

See also *SelLength* property, *SelText* property

SelText

SelText is the selected portion of the edit control's text.

```
property SelText: string;
```

Description Read *SelText* to determine the value of the selected text. Set *SelText* to replace the selected text with a new string. If there is no selection, but the edit control has focus, set *SelText* to insert a new string into the text at the cursor.

See also *ClearSelection* method, *GetSelTextBuf* method, *SelLength* property, *SelStart* property, *SetSelTextBuf* method, *Text* property

Events

The following table lists the events of *TCustomRichEdit*:

Event	Visibility	Documented in
OnProtectChange	protected	TCustomRichEdit
OnResizeRequest	protected	TCustomRichEdit
OnSaveClipboard	protected	TCustomRichEdit
OnSelectionChange	protected	TCustomRichEdit

OnProtectChange

OnProtectChange occurs when a user attempts to modify text that is marked as protected.

```
type TRichEditProtectChange =
  procedure(Sender: TObject; StartPos, EndPos: Integer; var AllowChange: Boolean) of object;

property OnProtectChange: TRichEditProtectChange;
```

Description Write an *OnProtectChange* event handler to allow modification of text with the protected attribute. The *StartPos* and *EndPos* parameters indicate the beginning and end of the range of text that would be affected by the modification, where 0 is the first character in the text. To allow modification of protected text, set *AllowChange* to *True*.

To set or unset the protected attribute for a section of text, select the text and use the *Protected* property of the rich edit control's *SelAttributes* property. To cause text to have the protected attribute by default, use the *Protected* property of the rich edit object's *DefAttributes* property.

See also *DefAttributes* property, *SelAttributes* property, *SelLength* property, *SelStart* property

OnResizeRequest

OnResizeRequest occurs when the text has grown either smaller or larger than the control's window size.

```
type
  TRichEditResizeEvent = procedure(Sender: TObject; Rect: TRect) of object;

property OnResizeRequest: TRichEditResizeEvent;
```

Description Write an *OnResizeRequest* event handler to respond to notifications that the edit control is trying to resize due to changes in the text. The *Rect* parameter gives the desired bounding rectangle that will display all the text, in screen coordinates.

See also *BoundsRect* property, *ClientRect* property, *Text* property

OnSaveClipboard

OnSaveClipboard occurs when the window of the edit control is about to be destroyed and the Clipboard contains information from the edit control.

```
type TRichEditSaveClipboard =
  procedure(Sender: TObject; NumObjects, NumChars: Integer; var SaveClipboard: Boolean)
    of object;
property OnSaveClipboard: TRichEditSaveClipboard;
```

Description Write an *OnSaveClipboard* event handler to respond to notifications that the window of the edit control is about to be destroyed and the Clipboard contains information from the edit control. The parameter *NumObjects* is the number of objects on the Clipboard, and *NumChars* is the number of characters on the Clipboard. To clear all information about the edit control from the Clipboard, set *SaveClipboard* to *False*.

See also *CopyToClipboard* method, *CutToClipboard* method

OnSelectionChange

OnSelectionChange occurs when the current selection changes.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
property OnSelectionChange: TNotifyEvent;
```

Description Write an *OnSelectionChange* event handler to respond to changes in the selected text. Changes in selection can occur when the user changes the selection using the mouse or keyboard, or when the selection is changed through the *SelText* property. Portions of the selection can be changed through the *Lines* property.

See also *AutoSelect* property, *ClearSelection* method, *Lines* property, *SelAttributes* property, *SelectAll* method, *SelLength* property, *SelStart* property, *SelText* property, *SetSelTextBuf* method

Methods

The following table lists the methods of *TCustomRichEdit*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject

Method	Visibility	Documented in
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomRichEdit
ClearSelection	public	TCustomEdit
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
CopyToClipboard	public	TCustomEdit
Create	public	TCustomRichEdit
CreateParams	protected	TCustomRichEdit
CreateParented	public	TWinControl
CreateWnd	protected	TCustomRichEdit
CutToClipboard	public	TCustomEdit
DefaultHandler	public	TObject
Destroy	public	TCustomRichEdit
DestroyComponents	public	TComponent
Destroying	public	TComponent
DestroyWnd	protected	TCustomRichEdit
DisableAlign	public	TWinControl
Dispatch	public	TObject
DoSetMaxLength	protected	TCustomRichEdit
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
FindText	public	TCustomRichEdit
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetSelLength	protected	TCustomRichEdit

Method	Visibility	Documented in
GetSelStart	protected	TCustomRichEdit
GetSelText	protected	TCustomRichEdit
GetSelTextBuf	public	TCustomRichEdit
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
PasteFromClipboard	public	TCustomEdit
Perform	public	TControl
Print	public	TCustomRichEdit
Realign	public	TWinControl
Refresh	public	TControl
RegisterConversionFormat	public	TCustomRichEdit
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
RequestSize	protected	TCustomRichEdit
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectAll	public	TCustomEdit
SelectionChange	protected	TCustomRichEdit
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetSelLength	protected	TCustomRichEdit
SetSelStart	protected	TCustomRichEdit
SetSelTextBuf	public	TCustomEdit

Method	Visibility	Documented in
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Clear

Clear deletes all text from the rich edit control.

```
procedure Clear; override;
```

Description Use *Clear* to replace the *Text* property with an empty string. *Clear* removes all text, not just the selected text. To remove just the selected text, use *ClearSelection*.

See also *ClearSelection* method, *Text* property

Create

Create creates an instance of *TCustomRichEdit*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to create an instance of *TCustomRichEdit* at runtime. For *TCustomRichEdit* objects placed on forms at design time, *Create* is called automatically.

Create overrides the inherited constructor to allocate helper objects that manage the rich text attributes of the text.

See also *TCustomMemo.Create* method

C

CreateParams

The *CreateParams* method initializes a window-creation parameter record passed in the *Params* parameter.

```
type
  TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;
```

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description The *CreateWnd* method calls *CreateParams* to initialize the parameters it passes to *CreateWindowHandle*. *TCustomRichEdit* overrides the inherited method to specify a Windows rich edit control.

See also *CreateWindowHandle* method, *CreateWnd* method

CreateWnd

The *CreateWnd* method creates a Windows control corresponding to the edit control.

```
procedure CreateWnd; override;
```

Description The *CreateWnd* method is called when the rich edit control is created, and whenever a change to the rich edit control requires the window to be recreated. *CreateWnd* first calls the inherited *CreateWnd*, and then sends the newly created window messages to request notification of Windows events, and, if the window is being recreated, reloads the rich text from a memory stream.

See also *CreateParams* method, *CreateWindowHandle* method, *DestroyWnd* method, *TWinControl.CreateWnd* method

Destroy

Destroy destroys an instance of *TCustomRichEdit*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TCustomRichEdit* object is not already freed and only then calls *Destroy*.

Destroy overrides the inherited method to destroy all the helper objects that were allocated to manage the rich text, before calling the inherited destructor.

See also *Free* method

DestroyWnd

The *DestroyWnd* method destroys the rich edit control's window handle.

```
procedure DestroyWnd; override;
```

Description Before the window handle is destroyed, *DestroyWnd* saves a copy of the control's text in a memory stream, frees any device contexts, and finally calls *DestroyWindowHandle*.

See also *CreateWnd* method, *DestroyWindowHandle* method

DoSetMaxLength

DoSetMaxLength sets the *MaxLength* property.

```
procedure DoSetMaxLength(Value: Integer); override;
```

Description *DoSetMaxLength* overrides the inherited method to implement the *MaxLength* property for a rich edit window. *DoSetMaxLength* sends an EM_EXLIMITTEXT message to the window of the rich edit control.

See also *MaxLength* property

FindText

FindText searches a given range in the text for a target string.

```
type
  TSearchType = (stWholeWord, stMatchCase);
  TSearchTypes = set of TSearchType;

function FindText(const SearchStr: string; StartPos, Length: Integer; Options: TSearchTypes):
  Integer;
```

Description Use *FindText* to search a range of text in a rich edit control for a string. The search will start at the position *StartPos*, where 0 is the first character. Use *Options* to specify whether the search should match whole words only and whether the search should be case sensitive. Use *FindText* rather than programmatically searching through the text of the control, to keep from being thrown off by the encoding of rich text characteristics.

See also *Text* property

GetSelLength

GetSelLength returns the *SelLength* property.

```
function GetSelLength: Integer; override;
```

Description *GetSelLength* overrides the inherited method to implement the *SelLength* property for a rich edit window. *GetSelLength* sends an EM_EXGETSEL message to the window of the edit control and returns the difference between the starting position and the ending position.

See also *SelLength* property, *SetSelLength* method

GetSelStart

GetSelStart returns the *SelStart* property.

```
function GetSelStart: Integer; override;
```

Description *GetSelStart* overrides the inherited method to implement the *SelStart* property for a rich edit window. *GetSelLength* sends an EM_EXGETSEL message to the window of the edit control and returns the starting position.

See also *SelStart* property, *SetSelStart* method

GetSelText

GetSelText returns the *SelText* property.

```
function GetSelText: string; override;
```

Description *GetSelText* overrides the inherited method to implement the *SelText* property for a rich edit window. *GetSelText* sends an EM_GETSELTEXT message to the window of the rich edit control.

See also *SelText* property

GetSelTextBuf

GetSelTextBuf copies the selected text into a buffer and returns the number of characters copied.

```
function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer; override;
```

Description Use *GetSelTextBuf* to copy the selected text into a character buffer. If there is no selection, the buffer receives an empty string. If the selection contains more than (*BufSize* – 1) characters, only the first (*BufSize* – 1) characters are copied. *GetSelTextBuf* returns the number of characters that were actually copied into the buffer.

See also *CopyToClipboard* method, *GetTextBuf* method, *SelLength* property, *SelText* property, *SetSelTextBuf* method

Print

Print formats and prints the contents of the rich edit control.

```
procedure Print(const Caption: string);
```

Description Use *Print* to print the contents of a rich edit control. The *Caption* parameter specifies the title that appears on the printed output.

See also *PageRect* property

RegisterConversionFormat

RegisterConversionFormat registers an association between a file extension and a *TConversion* object that can convert between the native file format and rich text format (RTF).

```
type TConversionClass = class of TConversion;
```

```
class procedure RegisterConversionFormat(const AExtension: string;
  AConversionClass: TConversionClass);
```

Description Rich edit controls encode text attributes in RTF format. To read from or write to files that encode text attributes in some other format, a converter is required. The converter is a descendant of *TConversion* that translates between the native file format and RTF when the text is streamed in or out. *RegisterConversionFormat* sets up an association between a file extension and a converter, so that all attempts to stream between a file with that extension and a rich edit control automatically invoke the appropriate converter.

Note Use the *Lines* property to stream the text to or from a file.

See also *DefaultConverter* property, *Lines* property, *PlainText* property, *TConversion* object

RequestSize

RequestSize responds to messages from the window of the rich text edit control when the text has grown either smaller or larger than the window size.

```
procedure RequestSize(const Rect: TRect); virtual;
```

Description *RequestSize* merely calls the *OnResizeRequest* event handler. *RequestSize* is exposed as a protected method so that derived classes can change the response to Windows messages about an impending window resize. When overriding *RequestSize*, include a call to the *OnResizeRequest* event handler.

See also *BoundsRect* property, *ClientRect* property, *OnResizeRequest* event, *Text* property

SelectionChange

SelectionChange responds to messages from window of the edit control that the selection has changed.

```
procedure SelectionChange; dynamic;
```

Description *SelectionChange* merely calls the *OnSelectionChange* event handler. *SelectionChange* is exposed as a protected method so that derived classes can change the response to Windows messages about selection changes. When overriding *SelectionChange*, include a call to the *OnSelectionChange* event handler.

See also *AutoSelect* property, *ClearSelection* method, *OnSelectionChange* event, *SelectAll* method, *SelAttributes* property, *SelLength* property, *SelStart* property, *SelText* property, *SetSelTextBuf* method

SetSelLength

SetSelLength sets the *SelLength* property.

procedure SetSelLength(Value: Integer); **override**;

Description *SetSelLength* overrides the inherited method to implement the *SelLength* property for a rich edit window. *SetSelLength* sends an EM_EXSETSEL message to the window of the edit control, specifying a selection that begins at *SelStart* and ends at *SelStart* + *Value*.

See also *GetSelLength* method, *SelLength* property, *SelStart* property

SetSelStart

SetSelStart returns the *SelStart* property.

procedure SetSelStart(Value: Integer); **override**;

Description *SetSelStart* overrides the inherited method to implement the *SelStart* property for a rich edit window. *SetSelLength* sends an EM_EXSETSEL message to the window of the edit control to position the cursor on the position indicated by *Value*.

See also *GetSelStart* method, *SelStart* property

TCustomServerSocket

Unit: ScktComp

TObject • TPersistent • TComponent • TCustomSocket • TCustomServerSocket

TCustomServerSocket is the base class for all server socket components.

Description *TCustomServerSocket* introduces new properties and events specific to the functions of a TCP/IP server.

Do not create instances of *TCustomServerSocket*. Use *TCustomServerSocket* as a base class when declaring server socket objects that listen for connection requests from other machines. To create a socket that responds to requests for connections from other machines, use *TServerSocket*.

See also *TServerWinSocket* object, *TServerSocket* object

Properties

The following table lists the properties of *TCustomServerSocket*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Name	published		TComponent
Owner	public, read-only		TComponent
ServerType	protected		TCustomServerSocket
Tag	published	0	TComponent
ThreadCacheSize	protected		TCustomServerSocket
VCLComObject	public		TComponent

ServerType

ServerType specifies whether each connection accepted by the server socket is non-blocking, or if it is automatically given a separate execution thread.

```
type TServerType = (stNonBlocking, stThreadBlocking);
```

```
property ServerType: TServerType;
```

Description Set *ServerType* to *stThreadBlocking* to automatically spawn a new thread for each socket connection accepted by the server socket. When *ServerType* is *stThreadBlocking*, the execution of the connection thread is suspended while reading or writing until all information has been transferred over the connection. The thread for each connection generates *OnClientRead* or *OnClientWrite* events when the server socket needs to read or write.

Set *ServerType* to *stNonBlocking* to handle all reading and writing over the socket connections asynchronously. When *ServerType* is *stNonBlocking*, all client connections are handled in a single execution thread by default. *OnClientRead* or *OnClientWrite* events occur when the client socket on the other end of one of the connections tries to send or receive information over the connection.

Use a non-blocking socket when the socket needs to synchronize reading and writing with client sockets.

Note When *ServerType* is *stThreadBlocking*, it is important that the *OnClient...* event handlers contain thread-safe code.

See also *OnClientRead* method, *OnClientWrite* method, *TServerClientThread* object

ThreadCacheSize

ThreadCacheSize is the maximum number of threads that can be reused for new client connections.

```
property ThreadCacheSize: Integer;
```

Description When *ServerType* is *stThreadBlocking*, each new connection that is accepted by the server socket is given a separate execution thread. In order to improve

performance, server sockets store the these threads in a cache rather than freeing them when the connection is closed. New connections can then reuse threads from the cache, rather than requiring the socket server to create a new thread every time a connection is accepted.

Set *ThreadCacheSize* to specify the number of threads that are cached for re-use. The ideal value for *ThreadCacheSize* depends on the number and frequency of client socket requests received by the server socket. If *ThreadCacheSize* is too low, the server socket will spend more time freeing and creating threads when client connections are accepted. If *ThreadCacheSize* is too high, the server socket may unnecessarily lock up the memory for threads that are never reused.

See also *ServerType* property, *TServerClientThread* object, *TServerWinSocket.ActiveThreads* property, *TServerWinSocket.IdleThreads* property

Events

The following table lists the events of *TCustomServerSocket*:

Event	Visibility	Documented in
OnClientConnect	protected	TCustomServerSocket
OnClientDisconnect	protected	TCustomServerSocket
OnClientError	protected	TCustomServerSocket
OnClientRead	protected	TCustomServerSocket
OnClientWrite	protected	TCustomServerSocket
OnGetSocket	protected	TCustomServerSocket
OnGetThread	protected	TCustomServerSocket
OnThreadEnd	protected	TCustomServerSocket
OnThreadStart	protected	TCustomServerSocket

OnClientConnect

OnClientConnect occurs when a client socket completes a connection accepted by the server socket.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;

property OnClientConnect: TSocketNotifyEvent;
```

Description Write an *OnClientConnect* event handler to take specific action when a client socket completes the socket connection to the server socket. For example, the socket may start reading or writing over the connection in an *OnClientConnect* event handler.

The order of server socket events leading up to *OnClientConnect* is as follows:

1. An *OnListen* event occurs just before the server socket is opened for listening.
2. The server socket receives client requests in a listening queue. The server socket accepts one of those requests, and receives a Windows socket handle for the new socket connection.
3. The server socket generates an *OnGetSocket* event, passing in the Windows socket handle. If a *TServerClientWinSocket* object for the server endpoint of the new connection is not created in an *OnGetSocket* event handler, the server socket creates one. The *TServerWinSocket* object continues to listen for other clients.
4. An *OnAccept* event occurs, using the new *TServerClientWinSocket* object.
5. If *ServerType* is *stThreadBlocking* and no thread is available in the cache, an *OnGetThread* event occurs. If the *OnGetThread* event handler does not create a thread, the server socket creates a *TServerClientThread*.
6. If *ServerType* is *stThreadBlocking*, an *OnThreadStart* event occurs as the thread begins execution.
7. The client completes the connection to the *TServerClientWinSocket* object and an *OnClientConnect* event occurs.

The *Sender* parameter is the *TServerWinSocket* object that describes the server endpoint of the listening connection. The *Socket* parameter is the *TServerClientWinSocket* object that describes the server endpoint of the connection to the client socket.

Note If *ServerType* is *stThreadBlocking*, make sure that all code in an *OnClientConnect* event handler is thread-safe. Use the *GetClientThread* method of the *Sender* parameter to access thread-specific information.

Note The *OnClientConnect* event handler for *TServerSocket* is also set when setting the *OnClientConnect* event handler of the associated *TServerWinSocket*.

See also *OnAccept* event, *OnGetSocket* event, *OnGetThread* event, *OnListen* event, *OnThreadStart* event, *ServerType* property, *TServerClientWinSocket* object, *TServerWinSocket* object, *TServerWinSocket.GetClientThread* method

OnClientDisconnect

OnClientDisconnect occurs when one of the connections to a client socket is closed.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;

property OnClientDisconnect: TSocketNotifyEvent;
```

Description Write an *OnClientDisconnect* event handler to take specific action when the connection to a client socket is ending. The termination of a client connection does not close the server socket. The server socket remains open and listening for client requests on its listening connection.

The *TServerClientWinSocket* that describes the server endpoint of the client connection is freed after *OnClientDisconnect*. If *ServerType* is *stThreadBlocking*, *OnThreadEnd* occurs after *OnClientDisconnect*.

The *Sender* parameter is the *TServerWinClient* object that describes the server endpoint of the listening connection. The *Socket* parameter is the *TServerClientWinSocket* object that describes the server endpoint of the connection with the client socket that is about to end.

Note If *ServerType* is *stThreadBlocking*, make sure that all code in an *OnClientDisconnect* event handler is thread-safe. Use the *GetClientThread* method of the *Sender* parameter to access thread-specific information.

Note The *OnClientDisconnect* event handler for *TServerSocket* is also set when setting the *OnClientDisconnect* event handler of the associated *TServerWinSocket*.

See also *OnThreadEnd* event, *ServerType* property, *TServerClientWinSocket* object, *TServerWinSocket* object, *TServerWinSocket.GetClientThread* method

OnClientError

OnClientError occurs when there is a failure in establishing, using, or terminating the socket connection to an individual client socket.

```
type
  TErrorEvent = (eeGeneral, eeSend, eeReceive, eeConnect, eeDisconnect, eeAccept);
  TSocketErrorEvent = procedure (Sender: TObject; Socket: TCustomWinSocket;
    ErrorEvent: TErrorEvent; var ErrorCode: Integer) of object;

property OnClientError: TSocketErrorEvent;
```

Description Write an *OnClientError* event handler to respond to errors that arise with the connection to a client socket. Set the *ErrorCode* parameter to 0 if the *OnClientError* event handler successfully deals with the error condition, to prevent an *ESocketError* from being raised.

The *Sender* parameter is the *TServerWinClient* object that describes the server endpoint of the listening socket. The *Socket* parameter is the *TServerClientWinSocket* object that describes the server endpoint of the connection to a client socket that encountered the error. The *ErrorEvent* parameter indicates what *Socket* was attempting to do when the error occurred. The *ErrorCode* parameter is the error code returned by the Windows socket API call.

Note If *ServerType* is *stThreadBlocking*, make sure that all code in an *OnClientError* event handler is thread-safe. Use the *GetClientThread* method of the *Sender* parameter to access thread-specific information.

Note The *OnClientError* event handler for *TServerSocket* is also set when setting the *OnClientError* event handler of the associated *TServerWinSocket*.

See also *ESocketError* object, *ServerType* property, *TServerClientWinSocket* object, *TServerWinSocket.GetClientThread* method

OnClientRead

OnClientRead occurs when the server socket should read information from a client socket.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;
property OnClientRead: TSocketNotifyEvent;
```

Description Write an *OnClientRead* event handler to read from the socket connection. If the *ServerType* property is *stThreadBlocking*, use a *TWinSocketStream* object to prevent problems that arise while reading from causing the execution thread to hang indefinitely. Otherwise, use the methods of the *Socket* parameter to perform the actual reading.

The *Sender* parameter is the *TServerWinClient* object that describes the server endpoint of the listening connection. The *Socket* parameter is the *TServerClientWinSocket* object that describes the server endpoint of the connection to the client socket that is sending the information.

Note If *ServerType* is *stThreadBlocking*, make sure that all code in an *OnClientRead* event handler is thread-safe. Use the *GetClientThread* method of the *Sender* parameter to access thread-specific information.

See also *ServerType* property, *TCustomWinSocket.ReceiveBuf* method, *TCustomWinSocket.ReceiveText* method, *TServerClientWinSocket* object, *TServerWinSocket.GetClientThread* method, *TWinSocketStream* object

OnClientWrite

OnClientWrite occurs when the server socket should write information to a client socket.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;
property OnClientWrite: TSocketNotifyEvent;
```

Description Write an *OnClientWrite* event handler to write to the socket connection. If the *ServerType* property is *stThreadBlocking*, use a *TWinSocketStream* object to prevent problems that arise while writing from causing the execution thread to hang indefinitely. Otherwise, use the methods of the *Socket* parameter to perform the actual writing.

The *Sender* parameter is the *TServerWinClient* object that describes the server endpoint of the listening connection. The *Socket* parameter is the *TServerClientWinSocket* object that describes the server endpoint of the connection to the client socket that will receive the information.

Note If *ServerType* is *stThreadBlocking*, make sure that all code in an *OnClientWrite* event handler is thread-safe. Use the *GetClientThread* method of the *Sender* parameter to access thread-specific information.

See also *ServerType* property, *TCustomWinSocket.SendBuf* method, *TCustomWinSocket.SendStream* method, *TCustomWinSocket.SendText* method, *TServerWinSocket.GetClientThread* method, *TWinSocketStream* object

OnGetSocket

OnGetSocket occurs when the server socket needs to create a new *TServerClientWinSocket* object to represent the server endpoint of a connection to a client socket.

```
type
  TSocket = u_int;
  TGetSocketEvent = procedure (Sender: TObject; Socket: TSocket;
    var ClientSocket: TServerClientWinSocket) of object;

property OnGetSocket: TGetSocketEvent;
```

Description Write an *OnGetSocket* event handler to create a specialized descendant of *TServerClientWinSocket* for the server socket component to use. Return the new *TServerClientWinSocket* object in the *ClientSocket* parameter. The *Sender* parameter is the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *TServerClientWinSocket* object, *TServerClientWinSocket.Create* method, *TServerWinSocket* object

OnGetThread

OnGetThread occurs when the server socket needs to create a new execution thread for a connection to a client socket.

```
type TGetThreadEvent = procedure (Sender: TObject; ClientSocket: TServerClientWinSocket;
  var SocketThread: TServerClientThread) of object;

property OnGetThread: TGetThreadEvent;
```

Description Write an *OnGetThread* event handler to create a specialized descendant of *TServerClientThread* for the connection to the client socket. Return the new *TServerClientThread* object in the *SocketThread* parameter. The *Sender* parameter is the *TServerWinSocket* object that describes the server endpoint of the listening connection. The *ClientSocket* parameter is the *TServerClientWinSocket* object that describes the server endpoint of the connection to the client socket that is about to be formed.

See also *ServerType* property, *TServerClientThread* object, *TServerClientWinSocket* object, *TServerWinSocket* object

OnThreadEnd

OnThreadEnd occurs when a client socket connection is terminated and the associated thread finishes execution.

```
type TThreadNotifyEvent = procedure (Sender: TObject; Thread: TServerClientThread) of object;
property OnThreadEnd: TThreadNotifyEvent;
```

Description Write an *OnThreadEnd* to take specific action when the thread for a client connection finishes execution. *OnThreadEnd* occurs after the *OnClientDisconnect* event.

The *Sender* parameter is the *TServerWinSocket* that describes the server endpoint of the listening connection. The *Thread* parameter is the thread that is finishing up.

See also *OnClientDisconnect* event, *TServerClientThread* event, *TServerWinSocket* object

OnThreadStart

OnThreadStart occurs when the execution thread for a connection to a client socket starts up.

```
type TThreadNotifyEvent = procedure (Sender: TObject; Thread: TServerClientThread) of object;
property OnThreadStart: TThreadNotifyEvent;
```

Description Write an *OnThreadStart* to initialize the *TServerClientThread* for the connection to a single client socket. The *Sender* parameter is the *TServerWinSocket* that describes the server endpoint of the listening connection. The *Thread* parameter is the thread that is about to start execution.

See also *TServerClientThread* object, *TServerWinSocket* object

Methods

The following table lists the methods of *TCustomServerSocket*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Close	public	TCustomSocket
Create	public	TComponent
DefaultHandler	public	TObject
Destroy	public	TCustomServerSocket
DestroyComponents	public	TComponent
Destroying	public	TComponent

Method	Visibility	Documented in
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetGetSocketEvent	protected	TCustomServerSocket
GetGetThreadEvent	protected	TCustomServerSocket
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetOnClientError	protected	TCustomServerSocket
GetOnClientEvent	protected	TCustomServerSocket
GetOnThreadEnd	protected	TCustomServerSocket
GetOnThreadStart	protected	TCustomServerSocket
GetParentComponent	public	TComponent
GetServerType	protected	TCustomServerSocket
GetThreadCacheSize	protected	TCustomServerSocket
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Open	public	TCustomSocket
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetGetSocketEvent	protected	TCustomServerSocket
SetGetThreadEvent	protected	TCustomServerSocket
SetOnClientError	protected	TCustomServerSocket
SetOnClientEvent	protected	TCustomServerSocket
SetOnThreadEnd	protected	TCustomServerSocket
SetOnThreadStart	protected	TCustomServerSocket
SetServerType	protected	TCustomServerSocket
SetThreadCacheSize	protected	TCustomServerSocket

Destroy

Destroy destroys an instance of *TCustomServerSocket*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the server socket is not already freed, and only then calls *Destroy*.

Destroy frees the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *Free* method, *TServerWinSocket* object

GetGetSocketEvent

GetGetSocketEvent returns the *OnGetSocket* event handler.

```
type TGetSocketEvent = procedure (Sender: TObject; Socket: TSocket;
  var ClientSocket: TServerClientWinSocket) of object;

function GetGetSocketEvent: TGetSocketEvent;
```

Description Applications cannot call this protected method. *GetGetSocketEvent* returns the *OnGetSocket* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *OnGetSocket* event, *TServerWinSocket.OnGetSocket* event

GetGetThreadEvent

GetGetThreadEvent returns the *OnGetThread* event handler.

```
type TGetThreadEvent = procedure (Sender: TObject; ClientSocket: TServerClientWinSocket;
  var SocketThread: TServerClientThread) of object;

function GetGetThreadEvent: TGetThreadEvent;
```

Description Applications cannot call this protected method. *GetGetThreadEvent* returns the *OnGetThread* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *OnGetThread* event, *TServerWinSocket.OnGetThread* event

GetOnClientError

GetOnClientError returns the *OnClientError* event handler.

```
type
  TErrorEvent = (eeGeneral, eeSend, eeReceive, eeConnect, eeDisconnect, eeAccept);
  TSocketErrorEvent = procedure (Sender: TObject; Socket: TCustomWinSocket;
   ErrorEvent: TErrorEvent; var ErrorCode: Integer) of object;

function GetOnClientError: TSocketErrorEvent;
```

Description Applications cannot call this protected method. *GetOnClientError* returns the *OnClientError* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *OnClientError* event, *TServerWinSocket.OnClientError* event

GetOnClientEvent

GetOnClientEvent returns the *OnClientEvent* event handler.

```
type TSocketEvent = (seLookup, seConnecting, seConnect, seDisconnect, seListen, seAccept,
    seWrite, seRead);

function GetOnClientEvent(Index: Integer): TSocketNotifyEvent;
```

Description Applications cannot call this protected method. *GetOnClientEvent* returns the *OnClientEvent* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *OnClientEvent* event, *TServerWinSocket.OnClientEvent* event

GetOnThreadEnd

GetOnThreadEnd returns the *OnThreadEnd* event handler.

```
type TThreadNotifyEvent = procedure (Sender: TObject; Thread: TServerClientThread) of object;

function GetOnThreadEnd: TThreadNotifyEvent;
```

Description Applications cannot call this protected method. *GetOnThreadEnd* returns the *OnThreadEnd* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *OnThreadEnd* event, *TServerWinSocket.OnThreadEnd* event

GetOnThreadStart

GetOnThreadStart returns the *OnThreadStart* event handler.

```
type TThreadNotifyEvent = procedure (Sender: TObject; Thread: TServerClientThread) of object;

function GetOnThreadStart: TThreadNotifyEvent;
```

Description Applications cannot call this protected method. *GetOnThreadStart* returns the *OnThreadStart* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *OnThreadStart* event, *TServerWinSocket.OnThreadStart* event

GetServerType

GetServerType returns the *ServerType* property.

```
type TServerType = (stNonBlocking, stThreadBlocking);
```

```
function GetServerType: TServerType;
```

Description Applications cannot call this protected method. *GetServerType* returns the value of the *ServerType* property of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *ServerType* property, *TServerWinSocket.ServerType* property

GetThreadCacheSize

GetThreadCacheSize returns the value of the *ThreadCacheSize* property.

```
function GetThreadCacheSize: Integer;
```

Description Applications cannot call this protected method. *GetThreadCacheSize* returns the value of the *ThreadCacheSize* property of the *TServerWinSocket* object that describes the server endpoint of the listening connection.

See also *ThreadCacheSize* property, *TServerWinSocket.ThreadCacheSize* property

SetGetSocketEvent

SetGetSocketEvent sets the *OnGetSocket* event handler.

```
type TGetSocketEvent = procedure (Sender: TObject; Socket: TSocket;
  var ClientSocket: TServerClientWinSocket) of object;

procedure SetGetSocketEvent(Value: TGetSocketEvent);
```

Description Applications cannot call this protected method. *SetGetSocketEvent* sets the *OnGetSocket* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *OnGetSocket* event.

See also *OnGetSocket* event, *TServerWinSocket.OnGetSocket* event

SetGetThreadEvent

SetGetThreadEvent sets the *OnGetThread* event handler.

```
type TGetThreadEvent = procedure (Sender: TObject; ClientSocket: TServerClientWinSocket;
  var SocketThread: TServerClientThread) of object;

procedure SetGetThreadEvent(Value: TGetThreadEvent);
```

Description Applications cannot call this protected method. *SetGetThreadEvent* sets the *OnGetThread* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *OnGetThread* event.

See also *OnGetThread* event, *TServerWinSocket.OnGetThread* event

SetOnClientError

SetOnClientError sets the *OnClientError* event handler.

```
type
  TErrorEvent = (eeGeneral, eeSend, eeReceive, eeConnect, eeDisconnect, eeAccept);
  TSocketErrorEvent = procedure (Sender: TObject; Socket: TCustomWinSocket;
   ErrorEvent: TErrorEvent; var ErrorCode: Integer) of object;

procedure SetOnClientError(Value: TSocketErrorEvent);
```

Description Applications cannot call this protected method. *SetOnClientError* sets the *OnClientError* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *OnClientError* event.

See also *OnClientError* event, *TServerWinSocket.OnClientError* event

SetOnClientEvent

SetOnClientEvent sets the *OnClientEvent* event handler.

```
type TSocketEvent = (seLookup, seConnecting, seConnect, seDisconnect, seListen, seAccept,
   seWrite, seRead);

procedure SetOnClientEvent(Index: Integer; Value: TSocketNotifyEvent);
```

Description Applications cannot call this protected method. *SetOnClientEvent* sets the *OnClientEvent* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *OnClientEvent* event.

See also *OnClientEvent* event, *TServerWinSocket.OnClientEvent* event

SetOnThreadEnd

SetOnThreadEnd sets the *OnThreadEnd* event handler.

```
type TThreadNotifyEvent = procedure (Sender: TObject; Thread: TServerClientThread) of object;

procedure SetOnThreadEnd(Value: TThreadNotifyEvent);
```

Description Applications cannot call this protected method. *SetOnThreadEnd* sets the *OnThreadEnd* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *OnThreadEnd* event.

See also *OnThreadEnd* event, *TServerWinSocket.OnThreadEnd* event

SetOnThreadStart

SetOnThreadStart sets the *OnThreadStart* event handler.

```
type TThreadNotifyEvent = procedure (Sender: TObject; Thread: TServerClientThread) of object;
```

```
procedure SetOnThreadStart(Value: TThreadNotifyEvent);
```

Description Applications cannot call this protected method. *SetOnThreadStart* sets the *OnThreadEnd* event handler of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *OnThreadEnd* event.

See also *OnThreadStart* event, *TServerWinSocket.OnThreadEnd* event

SetServerType

SetServerType sets the *ServerType* property.

```
type TServerType = (stNonBlocking, stThreadBlocking);
```

```
function GetServerType: TServerType;
```

Description Applications cannot call this protected method. *SetServerType* sets the *ServerType* property of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *ServerType* property.

See also *ServerType* property, *TServerWinSocket.ServerType* property

SetThreadCacheSize

SetThreadCacheSize sets the *ThreadCacheSize* property.

```
function GetThreadCacheSize: Integer;
```

Description Applications cannot call this protected method. *SetThreadCacheSize* sets the *ThreadCacheSize* property of the *TServerWinSocket* object that describes the server endpoint of the listening connection. Server socket objects use *TServerWinSocket* to implement their *ThreadCacheSize* property.

See also *ThreadCacheSize* property, *TServerWinSocket.ThreadCacheSize* property

C

TCustomSocket

Unit: ScktComp

TObject • TPersistent • TComponent • TCustomSocket

TCustomSocket is the abstract base class for all socket components.

Description *TCustomSocket* introduces properties, events, and methods to enable an application to work with Windows sockets. A Windows socket encapsulates a set of communication protocols to allow the application to connect to other machines for reading or writing information. Windows sockets provide connections based on the TCP/IP protocol. They also allow connections that use the Xerox Network System (XNS), Digital's DECnet protocol, or Novell's

IPX/SPX family. Sockets allow an application to form connections to other machines without being concerned with the details of the actual networking software.

Do not create instances of *TCustomSocket*. Use *TCustomSocket* as a base class when declaring socket objects that provide connections to other machines. Use a descendant of *TCustomSocket* to add the ability to communicate with other machines to an application. To create a socket that initiates the connection with another machine, use *TClientSocket*. To create a socket that responds to requests for connections from other machines, use *TServerSocket*.

See also *TClientSocket* object, *TCustomWinSocket* object, *TServerSocket* object

Properties

The following table lists the properties of *TCustomSocket*:

Property	Visibility	Default	Documented in
Active	protected		TCustomSocket
Address	protected		TCustomSocket
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Host	protected		TCustomSocket
Name	published		TComponent
Owner	public, read-only		TComponent
Port	protected		TCustomSocket
Service	protected		TCustomSocket
Tag	published	0	TComponent
VCLComObject	public		TComponent

Active

Active indicates whether the socket connection described by *TCustomSocket* is open and available for communication with other machines.

```
property Active: Boolean;
```

Description Before attempting to use or change the socket connection, read *Active* to determine whether the connection is open and ready. For client sockets, setting *Active* opens or shuts down a socket connection to another machine. For server sockets, setting *Active* opens or shuts down a listening connection that makes the socket available for client requests.

At design time, set *Active* to *True* to make the socket open a connection when the application starts running. At runtime, use the *Open* or *Close* method to open or close the connection.

See also *Close* method, *Open* method, *SetActive* method, *OnConnect* event, *OnDisconnect* event

Address

Address is the IP address of the server system.

property Address: string;

Description *Address* is a string of four numeric (byte) values in the standard internet dot notation, such as

```
123.197.1.2
```

For client sockets, set *Address* to the IP address of the server to which the socket should connect. When the connection is opened, the value of *Address* is bound to the connection.

If the socket specifies the *Host* property, the address for the connection is taken from the IP address associated with *Host*, rather than the value of *Address*. Specifying the server system by giving the IP address is faster, because the socket doesn't need to search for the IP address associated with the host name before it can locate the server system.

A single system may support more than one IP address.

Note Trying to change *Address* when the connection is open will raise an *ESocketError* exception.

See also *ESocketError* object, *Host* property, *SetAddress* method

C

Host

Host is an alias for the IP address of the server system.

property Host: string;

Description *Host* is a string containing the domain name and service of a particular system, such as

```
http://www.wSite.Com
```

For client sockets, set *Host* to the system with which the client socket should form a connection. When the socket opens a connection, it looks up the IP address for the server socket using the value of *Host*.

Some servers change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If *Host* is set, it takes precedence over the *Address* property when looking up the address of the server.

Note Trying to change *Host* when the connection is open will raise an *ESocketError* exception.

See also *Address* property, *ESocketError* object, *SetHost* method

Port

Port is the ID number that identifies the server socket connection.

```
property Port: Integer;
```

Description *Port* numbers allow a single system, identified by the *Host* or *Address* property, to host multiple connections simultaneously. Many values of *Port* are associated by convention with a particular service such as ftp or http.

For server sockets, *Port* is the ID of the connection on which the server socket listens for client requests. Server sockets generally set *Port* to a predefined value which clients can use to initiate connections.

For client sockets, *Port* is the ID of the desired server connection. The value of *Port* is usually associated with the service the client wishes to use on the server application.

Note Trying to change *Port* when the connection is open will raise an *ESocketError* exception.

See also *Address* property, *ESocketError* object, *Host* property, *Service* property, *SetPort* method

Service

Service is the name of the service for which the socket connection is used.

```
property Service: string;
```

Description Use *Service* to identify the use of the connection. Windows provides a number of standard service names such as ftp, http, finger, or time. Servers can specify additional services and their associated ports in a SERVICES file. For more information, see the Microsoft documentation for Windows sockets.

Certain port numbers are reserved for specific values of service. Thus, *Service* provides a more meaningful way to specify the server *Port* to use for the socket connection. For server sockets, using *Service* rather than *Port* ensures that the server will listen for TCP/IP requests on the appropriate port.

Note Trying to change *Service* when the connection is open will raise an *ESocketError* exception.

See also *ESocketError* object, *Port* property, *SetService* method, *TCustomWinSocket.LookupService* method

Events

The following table lists the events of *TCustomSocket*:

Event	Visibility	Documented in
OnAccept	protected	TCustomSocket
OnConnect	protected	TCustomSocket
OnConnecting	protected	TCustomSocket
OnDisconnect	protected	TCustomSocket
OnError	protected	TCustomSocket
OnListen	protected	TCustomSocket
OnLookup	protected	TCustomSocket
OnRead	protected	TCustomSocket
OnWrite	protected	TCustomSocket

OnAccept

OnAccept occurs on server sockets just after the connection to a client socket is accepted.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;
```

```
property OnAccept: TSocketNotifyEvent;
```

Description Write an *OnAccept* event handler for a server socket to take specific action after the connection to a client socket has been accepted. This is the first point when information is available about the local port and IP address that will be used in the server endpoint of the connection. The port number, and possibly the IP address, will differ from those of the listening socket.

Server sockets open a socket connection for listening to establish a queue to hold client requests. After a request from the queue is accepted by the server socket, an *OnAccept* event occurs. After the server socket accepts the connection, the client socket completes the connection, and an *OnConnect* event occurs for the client socket.

The *Sender* parameter is the server socket component that accepts the connection. The *Socket* parameter is the *TCustomWinSocket* object that describes the server endpoint of the connection to the client socket that is about to form.

See also *OnConnect* event, *OnListen* event, *TCustomWinSocket* object

OnConnect

OnConnect occurs on client sockets just after the connection to the server is opened.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;
```

```
property OnConnect: TSocketNotifyEvent;
```

Description Write an *OnConnect* event handler for a client socket to take specific action after the connection to a server socket has been established. Depending on the service, this may be the point when the socket should start reading or writing over the connection.

When a client socket opens a connection, the following events occur:

1. An *OnLookup* event occurs prior to an attempt to locate the server socket.
2. The Windows socket is set up, and initialized for event notification.
3. An *OnConnecting* event occurs after the server socket is located.
4. The connection request is accepted by the server and completed by the client socket.
5. An *OnConnect* event occurs after the connection is established.

The *Sender* parameter is the client socket component that initiates the connection. The *Socket* parameter is the *TCustomWinSocket* object that describes the client endpoint of the newly formed connection.

See also *OnAccept* event, *OnConnecting* event, *OnLookup* event, *TCustomWinSocket* object

OnConnecting

OnConnecting occurs for a client socket after the server socket has been located, but before the connection is established.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;
property OnConnecting: TSocketNotifyEvent;
```

Description Write an *OnConnecting* event handler for a client socket to take specific action just before the connection to a server socket is established. This is the first opportunity to obtain the actual port and IP address of the server endpoint of the connection that is about to form.

When a client socket opens a connection, the following events occur:

1. An *OnLookup* event occurs prior to an attempt to locate the server socket.
2. The Windows socket is set up, and initialized for event notification.
3. An *OnConnecting* event occurs after the server socket is located.
4. The connection request is accepted by the server and completed by the client socket.
5. An *OnConnect* event occurs after the connection is established.

The *Sender* parameter is the client socket component that initiates the connection. The *Socket* parameter is the *TCustomWinSocket* object that describes the server endpoint of the connection that is about to form.

See also *OnConnect* event, *OnLookup* event, *TCustomWinSocket* object

OnDisconnect

OnDisconnect occurs just before a client socket closes the connection to a server socket.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;

property OnDisconnect: TSocketNotifyEvent;
```

Description Write an *OnDisconnect* event handler to take specific action when the connection to a server socket is about to be terminated. *OnDisconnect* occurs after the value of the *Active* property is set to *False*, but before the actual connection is closed.

The *Sender* parameter is the client socket component. The *Socket* parameter is the *TCustomWinSocket* object that describes the client endpoint of the connection that is about to end.

See also *Active* property, *Close* method, *OnConnect* event

OnError

OnError occurs when the socket fails in making, using, or shutting down a connection.

```
type
  TErrorEvent = (eeGeneral, eeSend, eeReceive, eeConnect, eeDisconnect, eeAccept);
  TSocketErrorEvent = procedure (Sender: TObject; Socket: TCustomWinSocket;
   ErrorEvent: TErrorEvent; var ErrorCode: Integer) of object;

property OnError: TSocketErrorEvent;
```

Description Write an *OnError* event handler to respond to errors that arise with the socket connection. Set the *ErrorCode* parameter to 0 if the *OnError* event handler successfully deals with the error condition, to prevent an *ESocketError* from being raised.

The *Sender* parameter is the socket component. The *Socket* parameter is the *TCustomWinSocket* object that describes the local endpoint of the connection that gave rise to the error. The *ErrorEvent* parameter indicates what *Socket* was attempting to do when the error occurred. The *ErrorCode* parameter is the error code returned by the Windows socket API call.

See also *ESocketError* object, *TCustomWinSocket* object

OnListen

OnListen occurs just before a server socket is opened for listening.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;

property OnListen: TSocketNotifyEvent;
```

Description Write an *OnListen* event handler for a server socket to take specific action just before the socket is opened for listening. *OnListen* occurs after *Address* and *Port* have been bound to the socket connection, but before it is opened. This is the last opportunity to make changes to the socket endpoint before it is opened for listening. Use the *SocketHandle* property of the *Socket* parameter for Windows API calls that change the socket.

Server sockets open a socket connection for listening to establish a queue to hold client requests. The connection to a client socket is completed when a request from the queue is accepted.

The *Sender* parameter is the socket component that listens using the connection. The *Socket* parameter is the *TCustomWinSocket* object that describes the server endpoint of the listening connection that is about to open.

See also *Accept* event, *Address* property, *Port* property, *TCustomWinSocket* object

OnLookup

OnLookup occurs when a client socket is about to look up the server socket with which it wants to connect.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;

property OnLookup: TSocketNotifyEvent;
```

Description Write an *OnLookup* event handler for a client socket to take specific action just before attempting to locate a server socket. This is the first opportunity to make Windows API calls that affect the client properties of the socket, such as specifying a particular port number. Use the *SocketHandle* property of the *Socket* parameter for Windows API calls.

When a client socket opens a connection, the following events occur:

1. An *OnLookup* event occurs prior to an attempt to locate the server socket.
2. The Windows socket is set up, and initialized for event notification.
3. An *OnConnecting* event occurs after the server socket is located.
4. The connection request is accepted by the server and completed by the client socket.
5. An *OnConnect* event occurs after the connection is established.

The *Sender* parameter is the client socket component that initiates the connection. The *Socket* parameter is the *TCustomWinSocket* object that describes the client endpoint of the connection that will be formed with the server.

Note Changing the *Address*, *Host*, *Port*, or *Service* properties of the *Sender* parameter in an *OnLookup* event handler will have no effect on the address or port used to locate a server socket. These properties must be correct before the *Open* method is called.

See also *Address* property, *Host* property, *OnConnect* event, *OnConnecting* event, *Open* method, *Port* property, *Service* property, *TCustomWinSocket* object

OnRead

OnRead occurs when a client socket should read information from the socket connection.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;

property OnRead: TSocketNotifyEvent;
```

Description Write an *OnRead* event handler to read from the socket connection. The *Sender* parameter is the client socket component that initiated the connection. The *Socket* parameter is the *TCustomWinSocket* object that describes the client endpoint of the socket connection. If the socket is a blocking socket, use a *TWinSocketStream* object to read from the connection. Otherwise, use the methods of the *Socket* parameter to perform the actual reading.

See also *TCustomWinSocket* object, *TCustomWinSocket.ReceiveBuf* method, *TCustomWinSocket.ReceiveText* method, *TWinSocketStream* object

OnWrite

OnWrite occurs when a client socket should write information to the socket connection.

```
type TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;

property OnWrite: TSocketNotifyEvent;
```

Description Write an *OnWrite* event handler to write to the socket connection. The *Sender* parameter is the client socket component that initiated the connection. The *Socket* parameter is the *TCustomWinSocket* object that describes the client endpoint of the socket connection. If the socket is a blocking socket, use a *TWinSocketStream* object to write to the connection. Otherwise, use the methods of the *Socket* parameter to perform the actual writing.

See also *TCustomWinSocket.SendBuf* method, *TCustomWinSocket.SendStream* method, *TCustomWinSocket.SendStreamThenDrop* method, *TCustomWinSocket.SendText* method, *TWinSocketStream* object

Methods

The following table lists the methods of *TCustomSocket*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject

Method	Visibility	Documented in
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Close	public	TCustomSocket
Create	public	TComponent
DefaultHandler	public	TObject
Destroy	public	TComponent
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
DoActivate	protected	TCustomSocket
Error	protected	TCustomSocket
Event	protected	TCustomSocket
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
Loaded	protected	TCustomSocket
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Open	public	TCustomSocket
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetActive	protected	TCustomSocket
SetAddress	protected	TCustomSocket
SetHost	protected	TCustomSocket
SetPort	protected	TCustomSocket
SetService	protected	TCustomSocket

Close

Close shuts down the socket connection.

```
procedure Close;
```

Description Call *Close* to shut down the socket connection. *Close* sets the *Active* property to *False*.

For client sockets, *Close* terminates the connection to the server. For server sockets, *Close* shuts down the connection so that the server socket is no longer listening for client requests.

See also *Active* property, *OnDisconnect* event

DoActivate

DoActivate is the abstract prototype of a method to open or close the socket connection.

```
procedure DoActivate(Value: Boolean); virtual; abstract;
```

Description Descendants of *TCustomSocket* override *DoActivate* to provide a method that initiates or terminates the socket connection. The steps that need to be taken to open or close a connection depend on whether the socket is a client socket or a server socket.

See also *Active* property, *Open* method, *Close* method

Error

Error dispatches error events to an error event handler if it is assigned.

```
type TErrorEvent = (eeGeneral, eeSend, eeReceive, eeConnect, eeDisconnect, eeAccept);
```

```
procedure Error(Socket: TCustomWinSocket; ErrorEvent: TErrorEvent; var ErrorCode: Integer);
  virtual;
```

Description Applications cannot call *Error*. It is exposed as a protected method so that helper objects within the *ScktComp* unit can transmit errors to the socket component. Override *Error* to add class-specific error processing or to block the *OnError* event.

See also *OnError* event

Event

Event dispatches the events that occur in the normal operation of the socket to the appropriate event handlers.

```
type TSocketEvent = (seLookup, seConnecting, seConnect, seDisconnect, seListen, seAccept,
    seWrite, seRead);
```

```
procedure Event(Socket: TCustomWinSocket; SocketEvent: TSocketEvent); virtual;
```

Description Applications cannot call *Event*. It is exposed as a protected method so that helper objects within the *ScktComp* unit can invoke the appropriate event handlers during the normal operation of the socket. Override *Event* to introduce class-specific responses when certain events occur or to prevent the socket from calling the associated event handler.

See also *OnAccept* event, *OnConnect* event, *OnConnecting* event, *OnDisconnect* event, *OnListen* event, *OnLookup* event, *OnRead* event, *OnWrite* event

Loaded

Loaded opens or closes the socket connection when the socket is loaded from a file, depending on the saved value of the *Active* property.

```
procedure Loaded; override;
```

Description Applications cannot call this protected method. *Loaded* is called automatically after the socket component has been loaded from a stream. *Loaded* overrides the default method to implement the value of the *Active* property that was saved with the socket component.

See also *Active* property

Open

Open opens the socket connection.

```
procedure Open;
```

Description Call *Open* to initiate the socket connection. *Open* sets the *Active* property to *True*.

For client sockets, *Open* locates and connects to a server. For server sockets, *Open* opens the socket connection in a listening mode, but does not complete the connection to a client socket.

See also *Active* property, *OnConnect* event, *OnListen* event

SetActive

SetActive sets the *Active* property.

```
procedure SetActive(Value: Boolean);
```

Description *SetActive* is exposed as a protected method so that descendants of *TCustomSocket* can customize the implementation of the *Active* property. *SetActive* calls the *DoActivate* method to open or close the socket connection.

See also *Active* property, *DoActivate* method

SetAddress

SetAddress sets the *Address* property.

```
procedure SetAddress(Value: string);
```

Description *SetAddress* is exposed as a protected method so that descendants of *TCustomSocket* can customize the implementation of the *Address* property. *SetAddress* raises an *ESocketError* exception if an attempt is made to set the *Address* property when the socket connection is open.

See also *Active* property, *Address* property, *ESocketError* object

SetHost

SetHost sets the *Host* property.

```
procedure SetHost(Value: string);
```

Description *SetHost* is exposed as a protected method so that descendants of *TCustomSocket* can customize the implementation of the *Host* property. *SetHost* raises an *ESocketError* exception if an attempt is made to set the *Host* property when the socket connection is open.

See also *Active* property, *ESocketError* object, *Host* property

SetPort

SetPort sets the *Port* property.

```
procedure SetPort(Value: Integer);
```

Description *SetPort* is exposed as a protected method so that descendants of *TCustomSocket* can customize the implementation of the *Port* property. *SetPort* raises an *ESocketError* exception if an attempt is made to set the *Port* property when the socket connection is open.

See also *Active* property, *ESocketError* object, *Port* property

SetService

SetService sets the *Service* property.

```
procedure SetService(Value: string);
```

Description *SetService* is exposed as a protected method so that descendants of *TCustomSocket* can customize the implementation of the *Service* property. *SetService* raises an *ESocketError* exception if an attempt is made to set the *Service* property when the socket connection is open.

See also *Active* property, *ESocketError* object, *Service* property

TCustomStaticText

Unit: StdCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomStaticText

TCustomStaticText is the abstract base class for *TStaticText*.

Description *TCustomStaticText* is the base class for *TStaticText*, a windowed control that displays text on a form. This component functions like *TLabel*, except that it has a window handle.

TCustomStaticText cannot receive focus or contain other controls.

See also *TLabel* object, *TStaticText* object

Properties

The following table lists the properties of *TCustomStaticText*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
Alignment	protected	taLeftJustify	TCustomStaticText
AutoSize	protected	True	TCustomStaticText
BorderStyle	protected	sbsNone	TCustomStaticText
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
FocusControl	protected		TCustomStaticText

Property	Visibility	Default	Documented in
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ShowAccelChar	protected	True	TCustomStaticText
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Alignment

Determines how text is aligned within the component.

property Alignment: TAlignment;

Description The possible values of *Alignment* are

Value	Meaning
taLeftJustify	Text is aligned at the left of the control.
taCenter	Text is centered in the control.
taRightJustify	Text is aligned at the right of the control.

See also *TAlignment* type

Autosize

Determines whether the component is automatically resized to accommodate its text.

property AutoSize: Boolean;

Description If *Autosize* is *True*, the component is automatically resized—at runtime and design time—whenever the font size or *Caption* text changes.

See also *Caption* property, *Font* property

BorderStyle

Determines the appearance of the control's border.

```
type TStaticBorderStyle = (sbsNone, sbsSingle, sbsSunken);
property BorderStyle: TStaticBorderStyle;
```

Description The possible values of *BorderStyle* are

Value	Meaning
sbsNone	No visible border.
sbsSingle	Single-line border.
sbsSunken	Raised border.

FocusControl

Links the component with another windowed control.

```
property FocusControl: TWinControl;
```

Description Use the *FocusControl* property to link the *TCustomStaticText* instance with another windowed control. If the text of *Caption* includes an accelerator key, the component specified by *FocusControl* receives focus when the accelerator is pressed.

See also *Caption* property, *TLabel.FocusControl* property, *TWinControl* object

ShowAccelChar

Enables accelerator characters.

```
property ShowAccelChar: Boolean;
```

Description If *ShowAccelChar* is *True*, an ampersand in the component's *Caption* text appears as an underline and makes the character after it into an accelerator key. If *ShowAccelChar* is *False*, ampersands appear as ampersands.

See also *Caption* property

Methods

The following table lists the methods of *TCustomStaticText*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl

Method	Visibility	Documented in
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomStaticText
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent

Method	Visibility	Documented in
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Creates and initializes a *TCustomStaticText* instance.

constructor Create(AOwner: TComponent); **override**;

Description *Create* calls the inherited constructor method, then initializes the *TCustomStaticText* instance by adding the following flags to *ControlStyle*: *csCaptureMouse, csClickEvents, csSetCaption, csOpaque, csReplicatable, csDoubleClicks*. Create also sets *AutoSize* and *ShowAccelChar* to *True*.

See also *AutoSize* property, *ControlStyle* property, *ShowAccelChar* property

TCustomTabControl

Unit: ComCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomTabControl

TCustomTabControl is the base type for tab-control components such as *TTabControl* and *TPageControl*.

Description Use *TCustomTabControl* as a base class when defining a control that includes a set of notebook style tabs across the top. *TCustomTabControl* introduces many properties, methods, and events to manage appearance and behavior of the tabs, and to respond when the user selects one of the tabs.

See also *TPageControl* object, *TTabControl* object

Properties

The following table lists the properties of *TCustomTabControl*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
DisplayRect	protected, read-only		TCustomTabControl
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl

Property	Visibility	Default	Documented in
Hint	published		TControl
HotTrack	protected	False	TCustomTabControl
Left	published		TControl
MultiLine	protected	False	TCustomTabControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ScrollOpposite	protected	False	TCustomTabControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabHeight	protected	0	TCustomTabControl
TabIndex	protected	–1	TCustomTabControl
TabOrder	public	–1	TWinControl
TabPosition	protected	tpTop	TCustomTabControl
Tabs	protected		TCustomTabControl
TabStop	public	True	TWinControl
TabWidth	protected	0	TCustomTabControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

DisplayRect

DisplayRect is the bounding rectangle of a tab control client area.

property DisplayRect: TRect;

Description Use *DisplayRect* to obtain the coordinates, in pixels, of the portion of the tab control that represents the contents of the tab pages. The appearance of this portion of the tab control does not change when different tabs are selected.

Use *DisplayRect* rather than the *ClientRect* property to determine the portion of the tab control that can contain other objects.

See also *ClientRect* property

HotTrack

HotTrack determines whether labels on the tab under the mouse are automatically highlighted.

property HotTrack: Boolean;

Description Set *HotTrack* to *True* to provide visual feedback to the user about which tab would be selected if the mouse button is clicked. *HotTrack* is especially useful when the tab control contains multiple rows of tabs.

See also *MultiLine* property

MultiLine

MultiLine determines whether the tabs can appear on more than one row.

```
property MultiLine: Boolean;
```

Description Use *MultiLine* to determine how the tabs are displayed. If *MultiLine* is *True*, the tabs are displayed on more than one row when the number of tabs exceeds the number that fits across the top of the tab control. How many rows is determined by how many tabs are in the tab control. If *MultiLine* is *False*, the tabs are displayed on one row only, and the user must scroll the displayed scroll arrows to view all the tabs.

See also *ScrollOpposite* property

ScrollOpposite

ScrollOpposite determines how the rows of tabs are scrolled in a multiline tab control.

```
property ScrollOpposite: Boolean;
```

Description Set *ScrollOpposite* to determine where previous rows of tabs in a multi-line tab control move when the user selects a tab in another row. When *ScrollOpposite* is *True*, previous rows of tabs are moved to the bottom of the tab control if *TabPosition* is *tpTop*, or to the top if *TabPosition* is *tpBottom*. When *ScrollOpposite* is *False*, previous rows of tabs are moved to the back of all other rows of tabs.

For example, in a tab control with three rows of tabs at the top, if the user selects a tab in the second row, *ScrollOpposite* determines where the first row moves. If *ScrollOpposite* is *True*, the second row now appears at the front (bottom) row on top of the tab control, followed by the third row behind (above) it. The first row moves to the bottom of the tab control. If *ScrollOpposite* is *False*, the first row moves to the back (top), so that the tab control displays the second row in front (at the bottom of the tab region), followed by the third row (now in the middle), followed by the first row.

Note Setting *ScrollOpposite* to *True* automatically sets the *MultiLine* property to *True*.

See also *MultiLine* property, *TabPosition* property

TabHeight

TabHeight specifies the height, in pixels, of the tabs in the tab control.

property TabHeight: Smallint;

Description Set *TabHeight* to control the height of the tabs that appear above the client region of the tab control. The *TabHeight* property is the vertical size in pixels of the individual tabs. If *TabHeight* is set to 0, the tabs automatically size themselves to fit their text.

See also *DisplayRect* property, *Tabs* property, *TabWidth* property

TabIndex

TabIndex identifies the selected tab on a tab control.

property TabIndex: Integer;

Description Read *TabIndex* to determine which tab was selected by the user. *TabIndex* is the index of the tab in the list of labels maintained by the *Tabs* property. The first (leftmost) tab has a *TabIndex* of 0, the next has 1, and so on. If no tabs are selected, *TabIndex* has a value of –1.

Set *TabIndex* to programmatically change the selected tab in the tab control.

See also *Tabs* property

TabPosition

TabPosition determines whether tabs appear at the top or bottom.

type TTabPosition = (tpTop, tpBottom);

property TabPosition: TTabPosition;

Description Set *TabPosition* to tpTop to display the tabs at the top of the tab control. Set *TabPosition* to *tpBottom* to display the tabs at the bottom. If the *ScrollOpposite* property is *True*, tabs can appear at both top and bottom, with the row containing the current tab and all subsequent rows on the side indicated by *TabPosition*.

See also *ScrollOpposite* property

Tabs

Tabs contains the list of text strings that label the tabs of the tab control.

property Tabs: TStrings;

Description Use *Tabs* to access the labels of the tabs in the tab control, or any objects associated with those tabs using the *Objects* property of the *Tabs* object. Add new strings to the *Tabs* property to add new tabs to the tab control. Rearrange the *Tabs* list to rearrange the tabs in the tab control.

See also *TabIndex* property

TabWidth

TabWidth specifies the horizontal size, in pixels, of the tabs in the tab control.

```
property TabWidth: Smallint;
```

Description Set *TabWidth* to control the width of the tabs that appear above the client region of the tab control. The *TabWidth* property is the horizontal size in pixels of the individual tabs. If *TabWidth* is set to 0, the tabs automatically size themselves to fit their text.

See also *TabHeight* property, *Tabs* property

Events

The following table lists the events of *TCustomTabControl*:

Event	Visibility	Documented in
OnChange	protected	TCustomTabControl
OnChanging	protected	TCustomTabControl

OnChange

OnChange occurs after a new tab is selected.

```
type TNotifyEvent = procedure (Sender: TObject);
property OnChange: TNotifyEvent;
```

Description Write an *OnChange* event handler to take specific action immediately after the selected tab changes. Use the *TabIndex* property to determine which tab is now selected. This is the opportunity to make any changes to the control that reflect the new state implied by the selected tag.

Before the value of *TabIndex* changes, an *OnChanging* event occurs.

See also *OnChanging* event, *TabIndex* property

OnChanging

OnChanging occurs immediately before a new tab is selected.

```
type TTabChangingEvent = procedure (Sender: TObject; var AllowChange: Boolean) of object;
property OnChanging: TTabChangingEvent;
```

Description Write an *OnChanging* event handler to take specific action immediately before the selected tab changes. Set the *AllowChange* parameter to *False* to prevent the change.

Use an *OnChanging* event handler to prevent the user from leaving a tab setting until certain conditions have been met. An *OnChanging* event handler

can also be used to save information about the current state of the tab control before it is changed by a new tab selection.

See also *OnChange* event, *TabIndex* property

Methods

The following table lists the methods of *TCustomTabControl*:

Method	Visibility	Documented in
AlignControls	protected	TCustomTabControl
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanChange	protected	TCustomTabControl
CanFocus	public	TWinControl
Change	protected	TCustomTabControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomTabControl
CreateParams	protected	TCustomTabControl
CreateParented	public	TWinControl
CreateWnd	protected	TCustomTabControl
DefaultHandler	public	TObject
Destroy	public	TCustomTabControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DestroyWnd	protected	TCustomTabControl
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent

Method	Visibility	Documented in
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl

Method	Visibility	Documented in
Update	public	TWinControl
UpdateControlState	public	TWinControl

AlignControls

AlignControls aligns the controls contained in the tab control.

```
procedure AlignControls(AControl: TControl; var Rect: TRect); override;
```

Description Call *AlignControls* to align any child controls within the client area of the tab control. AlignControls uses the *Align* property value of each child control to determine how to align it. The *AControl* parameter can be **nil**. If *AControl* is not **nil**, that control takes precedence in alignment over other, similarly-aligned controls.

AlignControls overrides the inherited method to use the value of *DisplayRect* for the *Rect* parameter.

See also *DisplayRect* property, *TControl.Align* property

CanChange

CanChange determines whether the selected tab can be changed.

```
function CanChange: Boolean;
```

Description *CanChange* is called automatically when an attempt is made to change the selected tab. *CanChange* generates an *OnChanging* event so that an *OnChanging* event handler can prevent the selection from changing. If there is no *OnChanging* event handler, *CanChange* returns *True*, allowing the change.

Override *CanChange* to specify additional processing before the *OnChanging* event handler executes or to block the *OnChanging* event.

See also *OnChanging* event

Change

Change generates an *OnChange* event.

```
procedure Change;
```

Description *Change* is called automatically when the selected tab changes. Override *Change* to specify additional processing before the *OnChange* event handler executes, or to block the *OnChange* event.

See also *OnChange* event

Create

Create creates and initializes an instance of *TCustomTabControl*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to instantiate a tab control at runtime. Tab controls placed on forms at design time are created automatically. *Create* calls the inherited *Create* method and sets the initial values for the tab control, including the creation of a tab list that contains the text strings for the tabs.

See also *TWinControl.Create* method

CreateParams

CreateParams sets up the window-creation parameters for the tab control.

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description Tab controls call *CreateParams* internally when the window for the tab control must be generated. *CreateParams* overrides the inherited method to specify a tab control window with the current property settings.

See also *CreateWnd* method

CreateWnd

CreateWnd creates the window for a tab control.

```
procedure CreateWnd; override;
```

Description *CreateWnd* is called automatically when the tab control's window is recreated. *CreateWnd* calls the inherited *CreateWnd* method, before assigning any saved tab text strings in internal storage.

See also *DestroyWnd* method

Destroy

Destroy destroys an instance of *TCustomTabControl*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, use *Free*. *Free* verifies that the tab control is not already freed, and only then calls *Destroy*. *Destroy* frees the tab list that holds the list of text strings for the tabs and calls the inherited destructor.

See also *Free* method

DestroyWnd

DestroyWnd destroys the window used by the tab control.

procedure DestroyWnd; **override**;

Description *DestroyWnd* is called automatically when the tab control is destroyed on when its window needs to be recreated. *DestroyWnd* saves a copy of the tab text strings in internal storage before calling the inherited method.

See also *CreateWnd* method

TCustomTreeView

Unit: ComCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomTreeView

TCustomTreeView is the abstract base type for tree-view components such as *TTreeView*.

Description Use *TCustomTreeView* as a base class when defining components that present information in a hierarchical structure with nodes that can be expanded or collapsed. Most of the properties defined in *TCustomTreeView* are protected so that you can choose whether to publish them in your own tree view components.

Do not create instances of *TCustomTreeView*. To add a tree view control to a form, use *TTreeView*.

See also *TTreeNode* object, *TTreeNodes* object, *TTreeView* object

Properties

The following table lists the properties of *TCustomTreeView*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BorderStyle	protected	bsSingle	TCustomTreeView
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent

Property	Visibility	Default	Documented in
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
DropTarget	public		TCustomTreeView
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
HideSelection	protected	True	TCustomTreeView
Hint	published		TControl
Images	protected		TCustomTreeView
Indent	protected		TCustomTreeView
Items	protected		TCustomTreeView
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
ReadOnly	protected	False	TCustomTreeView
Selected	public		TCustomTreeView
ShowButtons	protected	True	TCustomTreeView
ShowHint	public		TControl
Showing	public, read-only		TWinControl
ShowLines	protected	True	TCustomTreeView
ShowRoot	protected	True	TCustomTreeView
SortType	protected	stNone	TCustomTreeView
StateImages	protected		TCustomTreeView
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
TopItem	public		TCustomTreeView
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

BorderStyle

BorderStyle determines whether the tree view control has a border.

```
type TBorderStyle = bsNone..bsSingle;
```

```
property BorderStyle: TBorderStyle;
```

Description Set *BorderStyle* to specify whether the tree view control should be outlined with a single-line border. These are the possible values:

Value	Meaning
bsNone	No visible border
bsSingle	Single-line border

See also *BoundsRect* property

DropTarget

DropTarget specifies which item in the tree view appears as the target of a drag and drop operation.

```
property DropTarget: TTreeNode;
```

Description Read *DropTarget* to determine whether a node in the tree view is drawn as the target of a drag and drop operation. Set *DropTarget* when specifying a particular node in the tree view as the drop target of a dragged item.

Note When *DropTarget* is set, the application must still handle the actual logic of accepting the dragged object by the indicated node.

See also *TControl.EndDrag* method, *TTreeNode* object, *TTreeNode.DropTarget* property

HideSelection

HideSelection determines whether a selected node appears selected when the focus shifts to another control.

```
property HideSelection: Boolean;
```

Description Use *HideSelection* to specify whether the user is given visual feedback about the current selection in the tree view when it does not have focus. If *True*, the selected node is not visually distinct from other nodes until focus returns to the control. If *False*, the node always appears selected.

See also *Selected* property

Images

Images determines which image list is associated with the tree view.

```
property Images: TImageList;
```

Description Use *Images* to provide a customized list of bitmaps that can be displayed to the left of a node's label. Individual nodes specify the image from this list that should appear by setting their *ImageIndex* property.

See also *OnGetImageIndex* event, *StateImages* property, *TTreeNode.ImageIndex* property, *TTreeNode.SelectedIndex* property

Indent

Indent specifies the amount of indentation in pixels when a list of child nodes is expanded.

property Indent: Integer;

Description Use *Indent* to determine how far child nodes are indented from their parent nodes when the parent is expanded.

See also *FullExpand* method

Items

Items lists the individual nodes that appear in the tree view control.

property Items: TTreeNodes;

Description Individual nodes in a tree view are *TTreeNode* objects. These individual nodes can be accessed by using the *Items* property along with the item's index into the tree view. For example, to access the second item in the tree view, you could use the following code.

```
MyTreeNode := TreeView1.Items[1]
```

When setting this property at design time in the Object Inspector the TreeView Items Editor appears. Use the New Item and New SubItem buttons to add items to the tree view. Use the *Text* property to modify what text is displayed in the label of the item.

At runtime nodes can be added and inserted by using the *TTreeNodes* methods *AddChildFirst, AddChild, AddChildObjectFirst, AddChildObject, AddFirst, Add, AddObjectFirst, AddObject* and *Insert*.

Note Accessing tree view items by index can be time-intensive, particularly when the tree view contains many items. For optimal performance, try to design your application so that it has as few dependencies on the tree view's item index as possible.

See also *TTreeNodes* object, *TTreeNodes.TreeView* property

ReadOnly

ReadOnly determines whether the user can edit the node labels.

property ReadOnly: Boolean;

Description Use *ReadOnly* to specify whether the user can edit the nodes of the tree view. If *ReadOnly* is *True*, the user can expand and collapse nodes, but can't edit their labels. If *ReadOnly* is *False*, the user can edit the labels as well. The default value is *False*.

See also *TTreeNode.Text* property

Selected

Selected returns the selected node in the tree view.

```
property Selected: TTreeNode;
```

Description Read *Selected* to access the selected node of the tree view. If there is no selected node, the value of *Selected* is **nil**.

Set *Selected* to set a node in the tree view. When a node becomes selected, the tree view's *OnChanging* and *OnChanged* events occur. Also, if the specified node is the child of a collapsed parent item, the parent's list of child items is expanded to reveal the specified node. In this case, the tree views *OnExpanded* and *OnExpanding* events occur as well.

See also *OnChange* event, *OnChanging* event, *SelectedIndex* property

ShowButtons

ShowButtons specifies whether to display plus (+) and minus (-) buttons to the left side of each parent item.

```
property ShowButtons: Boolean;
```

Description If *ShowButtons* is *True*, a button will appear to the left of each parent item. The user can click the button to expand or collapse the child items as an alternative to double-clicking the parent item.

See also *ShowLines* property, *ShowRoot* property

ShowLines

ShowLines specifies whether lines that link child nodes to their corresponding parent nodes are displayed.

```
property ShowLines: Boolean;
```

Description If *ShowLines* is *True*, lines linking child nodes to their parent nodes are displayed. Nodes at the root of the hierarchy are not automatically linked. To link nodes at the root, the *ShowRoot* property must also be set to *True*.

See also *ShowButtons* property, *ShowRoot* property

ShowRoot

ShowRoot specifies whether lines connecting top-level nodes are displayed.

property ShowRoot: Boolean;

Description To show lines connecting top-level nodes to a single root, set the tree view's *ShowRoot* and *ShowLines* properties to *True*.

See also *ShowButtons* property, *ShowLines* property

SortType

SortType determines if and how the nodes in a tree view are automatically sorted.

property SortType: TSortType;

Description Once a tree view is sorted, the original hierarchy is lost. That is, setting the *SortType* back to *stNone* will not restore the original order of items. These are the possible values:

Value	Meaning
stNone	No sorting is done.
stData	The items are sorted when the Data object or SortType is changed.
stText	The items are sorted when the Caption or SortType is changed.
stBoth	The items are sorted when either the Data object, the Caption or SortType is changed

Optionally, the *OnCompare* event can be hooked to handle comparisons. The *OnCompare* event will be called to compare two nodes for sorting.

See also *AlphaSort* method, *OnCompare* event

C

StateImages

StateImages determines which image list to use for state images.

property StateImages: TImageList;

Description Use *StateImages* to provide a set of bitmaps that reflect the state of tree view nodes. The state image appears as an additional image to the left of the item's icon.

See also *Images* property, *TTreeNodes.StateIndex* property

TopItem

TopItem is the topmost node that appears in the tree view.

property TopItem: TTreeNode;

Description When *TopItem* is changed, the tree view scrolls vertically so that the specified node is topmost in the list view.

See also *GetFirstNode* method

Events

The following table lists the events of *TCustomTreeView*:

Event	Visibility	Documented in
OnChange	protected	TCustomTreeView
OnChanging	protected	TCustomTreeView
OnCollapsed	protected	TCustomTreeView
OnCollapsing	protected	TCustomTreeView
OnCompare	protected	TCustomTreeView
OnDeletion	protected	TCustomTreeView
OnEdited	protected	TCustomTreeView
OnEditing	protected	TCustomTreeView
OnExpanded	protected	TCustomTreeView
OnExpanding	protected	TCustomTreeView
OnGetImageIndex	protected	TCustomTreeView
OnGetSelectedIndex	protected	TCustomTreeView

OnChange

OnChange occurs whenever the selection has changed from one node to another.

```
type TTVChangedEvent = procedure(Sender: TObject; Node: TTreeNode) of object;

property OnChange: TTVChangedEvent;
```

Description The *TTVChangedEvent* type points to a method that is called when the selection has changed from one item to another (*OnChange* event) in the *TTreeView* component. The *Node* parameter is the node whose selection state has changed.

See also *OnChanging* event, *Selected* property

OnChanging

OnChanging occurs whenever the selection is about to be changed from one node to another.

```
type TTVChangingEvent = procedure(Sender: TObject; Node: TTreeNode; var AllowChange:
    Boolean) of object;

property OnChanging: TTVChangingEvent;
```

Description The *TTVChangingEvent* type points to a method that is called when the selection is about to be changed from one node to another. Set *AllowChange* to *False*, to prevent selection from moving to a new node. The *Node* parameter specifies the currently selected node.

See also *OnChange* event, *Selected* property

OnCollapsed

OnCollapsed occurs after a node has been collapsed.

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode) of object;

property OnCollapsed: TTVExpandedEvent;
```

Description Write an *OnCollapsed* event handler to respond after a node in the tree view collapses. The *Node* parameter is the node whose children are no longer visible.

See also *TTreeNode.Collapse* method, *FullCollapse* method

OnCollapsing

OnCollapsing occurs when a node is about to be collapsed.

```
type TTVCollapsingEvent = procedure(Sender: TObject; Node: TTreeNode; var AllowCollapse:
    Boolean) of object;

property OnCollapsing: TTVCollapsingEvent;
```

Description The *TTVCollapsingEvent* type points to a method that is called when a node is about to be collapsed. Set the *AllowCollapse* parameter to *False* to prevent the node specified by the *Node* parameter from being collapsed.

See also *TTreeNode.Collapse* method, *FullCollapse* method

C

OnCompare

OnCompare occurs when two nodes must be compared during a sort of the nodes in the tree view.

```
type TTVCompareEvent = procedure(Sender: TObject; Node1, Node2: TTreeNode; Data:
    Integer; var Compare: Integer) of object;

property OnCompare: TTVCompareEvent;
```

Description Write an *OnCompare* event handler to customize the sort order of the nodes in the tree view. Set the *Compare* parameter to a value less than 0 if *Node1* is less than *Node2*. Set *Compare* to 0 if *Node1* is equivalent to *Node2*, and set *Compare* to a value greater than 0 if *Node1* is greater than *Node2*. If an *OnCompare* event handler is not provided, tree view nodes are sorted alphabetically, based on their labels.

See also *AlphaSort* method, *CustomSort* method, *SortType* property

OnDeletion

OnDeletion occurs when a node in the tree view is deleted.

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode) of object;
property OnDeletion: TTVExpandedEvent;
```

Description Write an *OnDeletion* event handler to respond when a node is deleted from the tree view control.

See also *TTreeNode.Delete* method

OnEdited

OnEdited occurs after the user edits the *Text* property of a node.

```
type TTVEditedEvent = procedure(Sender: TObject; Node: TTreeNode; var S: string) of object;
property OnEdited: TTVEditedEvent;
```

Description Write an *OnEdited* event handler to respond to changes the user makes to the node labels. The *Node* parameter is the node whose label was edited. The *S* parameter is the new value of the node's *Text* property. The node's label can be changed in an *OnEdited* event handler before the user's edits are committed. This event can occur only if *ReadOnly* is set to *False*.

See also *OnEditing* event, *ReadOnly* property, *TTreeNode.Text* property

OnEditing

OnEditing occurs when the user starts to edit the *Text* property of a node.

```
type TTVEditingEvent = procedure(Sender: TObject; Node: TTreeNode; var AllowEdit: Boolean)
  of object;
property OnEditing: TTVEditingEvent;
```

Description Write an *OnEditing* event handler to determine whether the user is allowed to edit the label of a specific node in the tree view. Set the *AllowEdit* parameter to *False* to prevent the user from editing the node specified by the *Node* parameter. To disallow editing of all nodes in the tree view, use the *ReadOnly* property instead.

See also *OnEdited* event, *ReadOnly* property

OnExpanded

OnExpanded occurs after a node is expanded.

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode) of object;
property OnExpanded: TTVExpandedEvent;
```

Description Write an *OnExpanded* event handler to respond when a node in the tree view is expanded. The *Node* parameter specifies the node whose children are now displayed to the user.

See also *FullExpand* method, *OnExpanding* event, *TTreeNode.Expand* method

OnExpanding

OnExpanding occurs when a node is about to be expanded.

```
type TTVExpandingEvent = procedure(Sender: TObject; Node: TTreeNode; var AllowExpansion:
  Boolean) of object;

property OnExpanding: TTVExpandingEvent;
```

Description Write an *OnExpanding* event handler to determine whether a node can be expanded. Set the *AllowExpansion* parameter to *False* to prevent the node from expanding.

See also *FullExpand* method, *OnExpanded* event, *TTreeNode.Expand* method

OnGetImageIndex

OnGetImageIndex occurs when the tree view looks up the *ImageIndex* of a node.

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode) of object;

property OnGetImageIndex: TTVExpandedEvent;
```

Description Write an *OnGetImageIndex* event handler to change the image index for the particular node before it is drawn. For example, the bitmap of a node can be changed to indicate a different state for the node.

See also *Images* property, *OnGetSelectedIndex* event, *TTreeNode.ImageIndex* property

C

OnGetSelectedIndex

OnGetSelectedIndex occurs when the tree view looks up the *SelectedIndex* of a node.

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode) of object;

property OnGetSelectedIndex: TTVExpandedEvent;
```

Description Write an *OnGetSelectedIndex* event handler to change the selected image index of a node before it is drawn.

See also *Images* property, *OnGetImageIndex* event, *TTreeNode.SelectedIndex* property

Methods

The following table lists the methods of *TCustomTreeView*:

Method	Visibility	Documented in
AlphaSort	public	TCustomTreeView
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomTreeView
CreateParented	public	TWinControl
CustomSort	public	TCustomTreeView
DefaultHandler	public	TObject
Destroy	public	TCustomTreeView
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
FullCollapse	public	TCustomTreeView
FullExpand	public	TCustomTreeView
GetHitTestInfoAt	public	TCustomTreeView

Method	Visibility	Documented in
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetNodeAt	public	TCustomTreeView
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
IsEditing	public	TCustomTreeView
LoadFromFile	public	TCustomTreeView
LoadFromStream	public	TCustomTreeView
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
SaveToFile	public	TCustomTreeView
SaveToStream	public	TCustomTreeView
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl

Method	Visibility	Documented in
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

AlphaSort

AlphaSort sorts all nodes alphabetically by label in the tree view control.

```
function AlphaSort: Boolean;
```

Description Call *AlphaSort* to sort the nodes of the tree view. If successful, *AlphaSort* returns *True*. To have the tree view maintain all nodes in a sorted order (for example, when the user edits the labels), use the *SortType* property.

See also *CustomSort* method, *SortType* property, *TTreeNode.AlphaSort* method

Create

Create creates an instance of a *TTreeView* control.

```
constructor Create(AOwner: TComponent);
```

Description Call *Create* to instantiate a tree view at runtime. Tree view controls placed on forms at design time are created automatically. After calling the inherited constructor, *Create* initializes many properties the control, including setting *BorderStyle* to *bsSingle* and setting *ShowButtons*, *ShowRoot*, *ShowLines*, and *HideSelection* to *True*.

See also *BorderStyle* property, *HideSelection* **property,** *ShowButtons* property, *ShowLines* property, *ShowRoot* property

CustomSort

CustomSort sorts the nodes in the tree view into a customized sort order.

```
type TTVCompare = function(lParam1, lParam2, lParamSort: Longint): Integer stdcall;

function CustomSort(SortProc: TTVCompare; Data: Longint): Boolean;
```

Description Use *CustomSort* to sort the nodes of the tree view where the sort order is defined by the *SortProc* parameter. The *lParam1* and *lParam2* parameters of the sort procedure can be cast to *TTreeNode* objects are compared. The *lParamSort* parameter of the sort procedure is the value of *Data* parameter of *CustomSort*. The sort procedure should return a value less than 0 if *lParam1* should come before *lParam2*, should return 0 if the two values are equivalent, and should return a value greater than 0 if *lParam1* should follow *lParam2*.

If the *SortProc* parameter is **nil**, the default *AlphaSort* method is called.

For example, to order a tree view in reverse alphabetical order, provide a callback function such as *CustomSortProc* below, which calls the global *AnsiStrIComp* function and negates its return value.

```
function CustomSortProc(Node1, Node2: TTreeNode; Data: integer): integer; stdcall;
begin
  Result := -AnsiStrIComp(PChar(Node1.Text), PChar(Node2.Text));
end;
```

This procedure can then be used as a parameter to *CustomSort* to sort the nodes of the tree view:

```
TreeView1.CustomSort(@CustomSortProc, 0);
```

Note To have the tree view sort all nodes automatically (for example, when the user edits the labels), use the *SortType* property instead, and provide an *OnCompare* event handler.

See also *AlphaSort* method, *OnCompare* event, *SortType* property

Destroy

The *Destroy* method destroys the tree view control and is called by the *Delete* method.

destructor Destroy; **override**;

Description The *Delete* method should be used to delete the tree node rather than the *Destroy* method.

See also *Create* method

FullCollapse

FullCollapse collapses all the nodes within a tree view control.

procedure FullCollapse;

Description Call *FullCollapse* to hide all the nodes in the tree view except those at the first level.

See also *OnCollapsing* event, *OnCollapsed* event, *TTreeNode.Collapse* method

FullExpand

FullExpand expands all nodes within the tree view control.

Procedure FullExpand;

Description Call *FullExpand* to display all the nodes in the tree view. If *ShowButtons* is *True*, then all buttons will change from '+' to '–'.

See also *OnExpanded* event, *OnExpanding* event, *ShowButtons* property, *TTreeNode.Expand* method

GetHitTestInfoAt

GetHitTestInfoAt returns information about the location of a point relative to the client area of the tree view control.

```
function GetHitTestInfoAt(X, Y: Integer): THitTests;
```

Description

Call *GetHitTestInfoAt* to determine what portion of the tree view, if any, sits under the point specified by the *X* and *Y* parameters. For example, use *GetHitTestInfoAt* to provide feedback about how to expand or collapse nodes when the mouse is over the relevant portions of the tree view.

GetHitTestInfo returns a *THitTests* type. The possible return values of *GetHitTestInfoAt* are:

Value	Meaning
htAbove	The point is above the client area of the tree view control.
htBelow	The point is below the client area of the tree view control.
htNowhere	The point is in the client area of the tree view control but below the last item.
htOnItem	The point is on the bitmap or label associated with an item.
htOnButton	The point is on the button associated with an item.
htOnIcon	The point is on the bitmap associated with an item.
htOnIndent	The point is on the indentation associated with an item.
htOnLabel	The point is on the label (text) associated with an item.
htOnRight	The point is in the area to the right of an item.
htOnStateIcon	The point is on the state icon for a tree view item that is in a user-defined state.
htToRight	The point is to the right of the client area of the tree view control.
htToLeft	The point is to the left of the client area of the tree view control.

The following code checks a list of *htHitTest* types and adds the ones that occur when the mouse was pressed while on the tree view.

```
procedure TForm1.TreeView1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  MyHitTest : THitTests;
begin
  MyHitTest := TreeView1.GetHitTestInfoAt(X,Y);
  if htNowhere in MyHitTest then
    ListBox1.Items.Add('NoWhere');
  if htOnItem in MyHitTest then
    ListBox1.Items.Add('OnItem');
  if htOnButton in MyHitTest then
    ListBox1.Items.Add('OnButton');
  if htOnIndent in MyHitTest then
    ListBox1.Items.Add('OnIndent');
  if htOnLabel in MyHitTest then
    ListBox1.Items.Add('OnLabel');
  if htOnRight in MyHitTest then
```

```
    ListBox1.Items.Add('OnRight');
  end;
```

See also *GetNodeAt* method

GetNodeAt

GetNodeAt returns the node that is found at the specified position.

function GetNodeAt(X, Y: Integer): TTreeNode;

Description Call *GetNodeAt* to access the node at the position specified by the *X* and *Y* parameters. *X* and *Y* specify the position in pixels relative to the top left corner of the tree view. If there is no node at the location, *GetNodeAt* returns **nil**.

For example, the following code uses *GetNodeAt* to add a dragged node as a child of the node under then mouse when it is dropped.

```
procedure TForm1.TreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  AnItem: TTreeNode;
  AttachMode: TNodeAttachMode;
  HT: THitTests;
begin
  if TreeView1.Selected = nil then Exit;
  HT := TreeView1.GetHitTestInfoAt(X, Y);
  AnItem := TreeView1.GetNodeAt(X, Y);
  if (HT - [htOnItem, htOnIcon, htNowhere, htOnIndent] <> HT) then
  begin
    if (htOnItem in HT) or (htOnIcon in HT) then AttachMode := naAddChild
    else if htNowhere in HT then AttachMode := naAdd
    else if htOnIndent in HT then AttachMode := naInsert;
    TreeView1.Selected.MoveTo(AnItem, AttachMode);
  end;
end;
```

See also *Items* property

IsEditing

IsEditing indicates whether a node is currently being edited by the user.

function IsEditing: Boolean;

Description *IsEditing* returns *True* if any node label in the tree view is being edited.

See also *Items* method

LoadFromFile

LoadFromFile reads the file specified in *FileName* and loads the data into the tree view.

```
procedure LoadFromFile(const FileName: string);
```

Description Use the *LoadFromFile* method to retrieve tree view data from a file and load into a tree view.

See also *LoadFromStream* method, *SaveToFile* method

LoadFromStream

LoadFromStream reads tree view data from a stream and stores the contents in the tree view.

```
procedure LoadFromStream(Stream: TStream);
```

Description Use *LoadFromStream* to read the nodes of the tree view from the specified stream. For example, an application can save the information displayed in a tree view as the data in a Binary Large Object (BLOB) field. *LoadFromStream* can retrieve the data using a *TBlobStream* object.

See also *LoadFromFile* method, *SaveToStream* method, *TBlobStream* object

SaveToFile

SaveToFile saves a the tree view to the file specified in *FileName*.

```
procedure SaveToFile(const FileName: string);
```

Description Use the *SaveToFile* method to store tree view data to a text file. The nodes can later be reloaded from the file into a new tree view object using the *LoadFromFile* method.

See also *LoadFromFile* method, *SaveToStream* method

SaveToStream

SaveToStream writes the data in the tree view to the stream passed as the *Stream* parameter.

```
procedure SaveToStream(Stream: TStream);
```

Description Use the *SaveToStream* method to stream out tree view data. It can be streamed back in to another tree view object using the *LoadFromStream* method.

See also *LoadFromStream* method, *SaveToFile* method

TCustomUpDown

Unit: ComCtrl

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomUpDown

TCustomUpDown is the base type for up-down size controls such as *TUpDown*.

Description Use *TCustomUpDown* as a base class when defining up-down controls such as the arrows that appear in a spin box. Up-down controls allow users to change the size of a numerical value by clicking on arrow buttons. *TCustomUpDown* introduces several properties, events, and methods to associate an up-down control with another control in a form and to communicate the size changes specified by the user.

Do not create instances of *TCustomUpDown*. To put an up-down control in a form, use *TUpDown*.

See also *TUpDown* object

Properties

The following table lists the properties of *TCustomUpDown*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
AlignButton	protected	udRight	TCustomUpDown
ArrowKeys	protected	True	TCustomUpDown
Associate	protected		TCustomUpDown
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl

Property	Visibility	Default	Documented in
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Increment	protected	1	TCustomUpDown
Left	published		TControl
Max	protected	100	TCustomUpDown
Min	protected		TCustomUpDown
Name	published		TControl
Orientation	protected	udVertical	TCustomUpDown
Owner	public, read-only		TComponent
Parent	public		TControl
ParentWindow	public		TWinControl
Position	protected		TCustomUpDown
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Thousands	protected	True	TCustomUpDown
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl
Wrap	protected		TCustomUpDown

AlignButton

AlignButton specifies the position of the up-down control relative to its companion control.

```
type TUDAlignButton = (udLeft, udRight);
```

```
property AlignButton: TUDAlignButton;
```

Description Set *AlignButton* to indicate where to position the up-down control. The up-down control appears attached to the outer edge of the control specified by the *Associate* property. These are the possible values:

Value	Meaning
udLeft	Position the up-down control on the left edge of the companion control.
udRight	Position the up-down control on the right edge of the companion control.

The up-down control resizes itself to the height of the companion control.

See also *Associate* property

ArrowKeys

ArrowKeys determines whether the up-down control can receive input from the *Up* arrow and *Down* arrow keys.

```
property ArrowKeys: Boolean;
```

Description Use *ArrowKeys* to specify whether the up-down control responds to keyboard input. *ArrowKeys* is used only when the up-down control is associated with a companion control. When *ArrowKeys* is *True*, the up-down control responds to the *Up* arrow and *Down* arrow keys, even when its companion control has input focus. When *ArrowKeys* is *False*, the user must click the buttons of the up-down control with the mouse. To set a companion control use the *Associate* property.

Note Up-down controls can respond to only the *Up* and *Down* arrow keys, even when the value of *Orientation* is *udHorizontal*.

See also *Associate* property, *Orientation* property

C

Associate

Associate specifies a companion control for the up-down control.

```
property Associate: TWinControl;
```

Description Use *Associate* to link the up-down control to a companion control (sometimes called a buddy window). The up-down control automatically positions itself on the outer edge of the companion control and adjusts its height to match the companion control.

At design time the companion control does not move with the up-down control. To keep the up-down control and its companion control together at design time, either multi-select both controls before moving them, or first move the companion control and then reset the *Associate* property for the up-down control.

See also *AlignButton* property, *ArrowKeys* property, *Position* property

Increment

Increment is the amount the *Position* value changes each time the up or down button is pressed.

```
property Increment: Integer;
```

Description Use *Increment* to control the granularity of changes the up-down control can make. Set *Increment* to indicate the number of units between *Min* and *Max* the *Position* moves whenever the user clicks on one of the arrow buttons. *Increment* represents the mapping between the units used to specify *Min*, *Max*, and *Position*, and the logical units represented by the *up-down control*.

See also *Max* property, *Min* property, *Position* property

Max

Max is the maximum value of the *Position* property.

```
property Max: SmallInt;
```

Description Use *Max* to provide an upper limit to the range of values the *Position* property can assume.

See also *Min* property, *Position* property

Min

Min is the minimum value of the *Position* property.

```
property Min: SmallInt;
```

Description Use *Min* to provide a lower limit to the range of values the *Position* property can assume.

See also *Max* property, *Position* property

Orientation

Orientation specifies the direction that the arrow keys point.

```
type TUDOrientation = (udHorizontal, udVertical);
property Orientation: TUDOrientation;
```

Description Use *Orientation* to determine the orientation of the arrow buttons. These are the possible values:

Value	Meaning
udHorizontal	The arrow buttons point to the left and right, and appear side by side.
udVertical	The arrow buttons point up and down and are stacked vertically.

Note Even when the *Orientation* is *udHorizontal*, the up-down control can respond only to the *Up* and *Down* arrow keys.

See also *ArrowKeys* property

Position

Position is the current value represented by the up-down control.

property Position: SmallInt;

Description Use *Position* to get or change the value the up-down control represents. *Position* must be a value between *Min* and *Max*.

When the *Associate* property specifies an edit control, its text is automatically set to the value of *Position*. When *Position* changes, the text of the edit control is updated. If the *Associate* property is **nil**, use the *OnClick* event to respond to changes in the value of *Position*.

See also *Associate* property, *Max* property, *Min* property, *OnChanging* event, *Thousands* property

Thousands

Thousands determines whether a thousands separator appears between every three digits of a decimal string.

property Thousands: Boolean;

Description Use *Thousands* to control the formatting of the text of the companion control of the up-down control. If *Thousands* is *True*, a separator appears between every three digits. The numeric value that is displayed by the companion control is given by the *Position* property.

See also *Associate* property, *Position* property

Wrap

Wrap specifies how the up-down control handles attempts to exceed the value of *Max* or *Min*.

property Wrap: Boolean;

Description Use *Wrap* to specify whether the up-down control treats the range determined by the *Max* and *Min* properties as a continuous loop. If *Wrap* is *True*, the value of *Position* becomes the value of the *Min* property when the user increments *Position* beyond the value of *Max*, and becomes the value of *Max* when the user decrements *Position* beyond the value of *Min*.

Max property, *Min* property, *Position* property

Events

The following table lists the events of *TCustomUpDown*:

Event	Visibility	Documented in
OnChanging	protected	TCustomUpDown
OnClick	protected	TCustomUpDown

OnChanging

OnChanging occurs when the position of an up-down control is about to change.

```
type TUDChangingEvent = procedure (Sender: TObject; var AllowChange: Boolean) of object;

property OnChange: TNotifyEvent;
```

Description Write an *OnChanging* event handler to conditionally disallow changes to the value of *Position*. Set the *AllowChange* parameter to *False* to prevent a change to the current value of *Position*.

OnChanging occurs both when the user clicks the arrow buttons of the up-down control and when an attempt is made to change the value of *Position* programmatically. To determine which button the user clicked, use the *OnClick* event, which precedes the *OnChanging* event.

See also *CanChange* method, *OnClick* event

OnClick

OnClick occurs when the user clicks an arrow button.

```
type
  TUDBtnType = (btNext, btPrev);
  TUDClickEvent = procedure (Sender: TObject; Button: TUDBtnType) of object;

property OnClick: TUDClickEvent
```

Description Write an *OnClick* event handler to take specific action when the user clicks one of the arrow buttons. For example, use an *OnClick* event handler to track the direction in which *Position* is about to change so that an *OnChanging* event handler, which is called immediately afterward, can determine whether the change should be allowed. If the *Associate* property is not used to automatically update the text of an edit control when the value of *Position* changes, use the *OnClick* event handler to respond to changes just before the value of *Position* is changed.

The *Button* parameter specifies how the value of *Position* is about to change. When *Button* is *btNext*, the Up or Right arrow was clicked, and the value of *Position* is about to increase by *Increment*. When *Button* is *btPrev*, the Down or

Left arrow was clicked, and the value of *Position* is about to decrease by *Increment*.

See also *Associate* property, *Increment* property, *OnChanging* event, *Position* property

Methods

The following table lists the methods of *TCustomUpDown*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanChange	protected	TCustomUpDown
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Click	protected	TCustomUpDown
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomUpDown
CreateParams	protected	TCustomUpDown
CreateParented	public	TWinControl
CreateWnd	protected	TCustomUpDown
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject

Method	Visibility	Documented in
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TCustomUpDown
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl

Method	Visibility	Documented in
Update	public	TWinControl
UpdateControlState	public	TWinControl

CanChange

CanChange specifies whether the value of *Position* can be changed.

```
function CanChange: Boolean;
```

Description *CanChange* is called automatically when the user clicks the arrow buttons or the an attempt is made to change the value of *Position* programmatically. *CanChange* generates an *OnChanging* event, and allows the change unless it is prevented by an *OnChanging* event handler.

Override *CanChange* to prevent changes to the value of *Position* that are not limited by an *OnChanging* event handler, or to block the *OnChanging* event.

See also *OnChanging* event, *Position* property

Click

Click generates an *OnClick* event.

```
TUDBtnType = (btNext, btPrev);
procedure Click(Button: TUDBtnType); dynamic;
```

Description *Click* is called automatically when the user clicks an arrow button or presses an *Up* or *Down* arrow key. Override this method to specify special processing that should occur before the *OnClick* event handler executes, or to block the *OnClick* event. The *Button* parameter specifies which button was clicked: *btNext* refers to the Up or Right arrow, and *btPrev* refers to the Down or Left arrow.

See also *OnClick* event

Create

Create creates and initializes an instance of *TCustomUpDown*.

```
constructor Create(AOwner: TComponent); override;
```

Description *Call Create* to instantiate an up-down control at runtime. Up-down controls placed on forms at design time are created automatically.

After calling the inherited constructor, Create initializes the following properties:

- *Height* and *Width* are given values based on the size of scroll bar arrows.
- *ArrowKeys* is set to *True*.
- *Max* is set to 100 and *Increment* is set to 1.

- *AlignButton* is set to *udRight* and *Orientation* is set to *udVertical*.
- *Thousands* is set to *True*.

See also *AlignButton* property, *ArrowKeys* property, *Height* property, *Increment* property, *Max* property, *Orientation* property, *Thousands* property, *TWinControl.Create* method, *Width* property

CreateParams

CreateParams initializes the window creation parameters for the up-down control.

```
procedure CreateParams(var Params: TCreateParams); override;
```

Description *CreateParams* is called automatically before the window for an up-down control is created. *CreateParams* calls the inherited method, then sets the *Params* parameter to reflect the values of the *AlignButton*, *ArrowKeys*, *Orientation*, *Thousands*, and *Wrap* properties.

See also *AlignButton* property, *ArrowKeys* property, *Orientation* property, *Thousands* property, *TWinControl.CreateParams* method, *Wrap* property

CreateWnd

CreateWnd creates the window used by the up-down control.

```
procedure CreateWnd; override;
```

Description After calling the inherited method to create the up-down control's window, *CreateWnd* sends the newly created windows messages so that it reflects the current values of the *Associate*, *Increment*, *Max*, *Min*, and *Position* properties.

See also *Associate* property, *CreateParams* method, *Increment* property, *Max* property, *Min* property, *Position* property, *TWinControl.CreateWnd* method

Notification

Notification updates the *Associate* property when the control it references is destroyed.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description After calling the inherited method, *Notification* checks whether the control specified by the *Associate* property is about to be destroyed, and if so, sets the *Associate* property to **nil**.

See also *Associate* property

TCustomWebDispatcher

Unit: HTTPApp

TObject • TPersistent • TComponent • TDataModule • TCustomWebDispatcher

TCustomWebDispatcher is the abstract base class for Web dispatcher objects.

Description Add a descendant of *TCustomWebDispatcher* to a Web application to enable it to respond to HTTP request messages. When a Web application is created, it automatically includes a *TWebModule* object, which is a descendant of *TCustomWebDispatcher*. Keep the default Web module, or replace it with another data module that contains a descendant of *TCustomWebDispatcher*, such as *TWebDispatcher*.

TCustomWebDispatcher manages the action items that assemble responses to HTTP request messages. When the *TWebApplication* object receives an HTTP request message, *TCustomWebDispatcher* passes the request on to the appropriate action item or action items to service the message.

TCustomWebDispatcher also keeps track of information about the current HTTP message for the benefit of *TCustomContentProducer* objects.

See also *TCustomContentProducer* object, *TWebActionItem* object, *TWebActionItems* object, *TWebApplication* object, *TWebDispatcher* object, *TWebModule* object, *TWebRequest* object, *TWebResponse* object

Properties

The following table lists the properties of *TCustomWebDispatcher*:

Property	Visibility	Default	Documented in
Action	public, read-only		TCustomWebDispatcher
Actions	public		TCustomWebDispatcher
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
DesignOffset	public		TDataModule
DesignSize	public		TDataModule
Name	published		TComponent
Owner	public, read-only		TComponent
Request	public, read-only		TCustomWebDispatcher
Response	public, read-only		TCustomWebDispatcher

Property	Visibility	Default	Documented in
Tag	published	0	TComponent
VCLComObject	public		TComponent

Action

Action provides indexed access to every action item managed by the *TCustomWebDispatcher* object.

```
property Action[Index: Integer]: TWebActionItem;
```

Description Use *Action* to iterate over all of the actions in the *Actions* collection. Zero is the index of the first action item, one is the index of the second action item, and so on. To access individual actions by name, use the *ActionByName* method.

See also *ActionByName* method, *Actions* property, *TWebActionItem* object

Actions

The *Actions* property is the collection of action items managed by the *TCustomWebDispatcher* object.

```
property Actions: TWebActionItems;
```

Description Use the methods of *Actions* to add action items for use by the dispatcher, to clear the set of action items managed by the dispatcher, or to access the attributes of the action items.

See also *Action* property, *ActionByName* method, *TWebActionItems* object

Request

Request is the *TWebRequest* object that the dispatcher is currently handling.

```
property Request: TWebRequest;
```

Description Use *Request* to obtain information about the current HTTP request message. The dispatcher matches this *TWebRequest* object to an action item, and invokes the *OnAction* event handler of the action item to generate and send a response to the HTTP request message. *OnAction* event handlers often use descendants of *TCustomContentProducer* to help generate the response to the request message. These content producers can use the *Request* property to correctly produce the content of the HTTP response.

See also *Dispatch* method, *Response* property, *TCustomContentProducer* object, *TWebActionItem.OnAction* event

Response

Response is the *TWebResponse* object for the HTTP message that the dispatcher is currently handling.

```
property Response: TWebResponse;
```

Description Use *Response* to obtain information about the emerging response to the current HTTP request message. The dispatcher matches the request object specified by the *Request* property to an action item. It then invokes the *OnAction* event handler of the action item to fill in the *TWebResponse* object specified by the *Response* property. *OnAction* event handlers often use descendants of *TCustomContentProducer* to help generate the response. These content producers can use *Response* to determine the state of the emerging response that is being produced.

See also *Dispatch* method, *Request* property, *TCustomContentProducer* object, *TWebActionItem.OnAction* event

Events

The following table lists the events of *TCustomWebDispatcher*:

Event	Visibility	Documented in
AfterDispatch	protected	TCustomWebDispatcher
BeforeDispatch	protected	TCustomWebDispatcher
OnCreate	published	TDataModule
OnDestroy	published	TDataModule

C

AfterDispatch

AfterDispatch occurs after the HTTP response message has been successfully filled out by the action items but not yet sent.

```
type THTTPMethodEvent = procedure (Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean) of object;

property AfterDispatch: THTTPMethodEvent;
```

Description Write an *AfterDispatch* event handler to provide a final check of the HTTP response message before it is sent to the Web client. An *AfterDispatch* event handler can let a response stand as is, change the response, send the response, or change the *Handled* parameter to *False* to prevent the Web application from sending the response.

The *AfterDispatch* event does not occur if all of the action items indicate that the response was not handled, or if the response is sent by any of the action items.

See also *Actions* property, *BeforeDispatch* event, *TWebResponse* object

BeforeDispatch

BeforeDispatch occurs before the dispatcher tries to match the HTTP request message with any of its action items.

```
type THTTPMethodEvent = procedure (Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean) of object;

property BeforeDispatch: THTTPMethodEvent;
```

Description Write a *BeforeDispatch* event handler to preprocess all HTTP request messages. A *BeforeDispatch* event handler can enable or disable the action items appropriate to the response, begin filling out the response object, or provide any other necessary preprocessing.

If the *BeforeDispatch* event handler finishes filling out the response object, it should change the *Handled* parameter to *True* so the dispatcher will not send the request on to any of the action items. If the *BeforeDispatch* event handler sends the response message, the dispatcher will not pass the request on to any of the action items, even if the *Handled* parameter is left as *False*.

If the *BeforeDispatch* event handler sets the *Handled* parameter to *True* but does not send the response, the Web dispatcher will generate an *AfterDispatch* event.

See also *Actions* property, *AfterDispatch* event, *TWebResponse* object

Methods

The following table lists the methods of *TCustomWebDispatcher*:

Method	Visibility	Documented in
ActionByName	public	TCustomWebDispatcher
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TCustomWebDispatcher
CreateNew	public	TDataModule
DefaultHandler	public	TObject
Destroy	public	TCustomWebDispatcher
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
DispatchAction	protected	TCustomWebDispatcher

Method	Visibility	Documented in
DoAfterDispatch	protected	TCustomWebDispatcher
DoBeforeDispatch	protected	TCustomWebDispatcher
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TCustomWebDispatcher
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

ActionByName

ActionByName returns the action item identified by the *AName* parameter.

```
function ActionByName(const AName: string): TWebActionItem;
```

Description Call *ActionByName* to gain access to an individual action item when its name is known. Change the properties of the action item so that it handles different HTTP request messages, or invoke its *OnAction* event handler to respond to an HTTP request message.

See also *Action* property, *Actions* property, *TWebActionItem* object, *TWebActionItem.Name* property

Create

Create creates an instance of *TCustomWebDispatcher*.

```
constructor Create(AOwner: TComponent); override;
```

Description Applications should not call the constructor for a *TCustomWebDispatcher*. Web applications automatically include a Web module. If the Web module is replaced by another data model, the Web dispatcher should be added to the new data module at design time. Objects placed at design time are created automatically. If an application tries to create a dispatcher object in a Web module, or in a data module that already has a Web dispatcher, an exception is raised.

Create creates the *TWebActionItems* object to hold the action items managed by the dispatcher. If the Web dispatcher is included in another data module, it adds a reference to itself to any content producer in that data module.

Finally, when the Web dispatcher is created at runtime, *Create* generates an *OnCreate* event.

See also *Actions* property, *OnCreate* event, *TCustomContentProducer.Dispatcher* property

Destroy

Destroy frees an instance of *TCustomWebDispatcher*.

```
destructor Destroy; override;
```

Description Do not destroy the *TCustomWebDispatcher* directly in an application. The dispatcher is freed automatically when the server application shuts down.

Destroy frees the *TWebActionItems* object used to hold the collection of action items.

See also *Actions* property, *Free* method

DispatchAction

DispatchAction passes an HTTP request message to the action items in *Actions* until it is handled.

```
function DispatchAction(Request: TWebRequest; Response: TWebResponse): Boolean;
```

Description The Web application calls the *DispatchAction* method when it receives an HTTP request message, so that the dispatcher can dispatch the request to any action items that should generate the response.

Before *DispatchAction* looks for action items to handle the HTTP request message represented by the *Request* parameter, it generates a *BeforeDispatch* event. If the *BeforeDispatch* event handler handles the action, the dispatcher does not pass the request message to any of the action items.

If there is no *BeforeDispatch* event handler, or of the *BeforeDispatch* event handler does not handle the request, *DispatchAction* attempts to match the *MethodType* and *PathInfo* of the request with the *MethodType* and *PathInfo* properties of each action item in the *Actions* property. When an action item matches the *MethodType* and *PathInfo* of the request, *DispatchAction* calls its *OnAction* event handler.

The *OnAction* event handler can complete the response to the request, or indicate that the request is not fully handled and allow another action item to finish processing the request. If the *OnAction* event handler does not complete the request, *DispatchAction* continues checking the action items in the *Actions* property and calling the *OnAction* event handler when a match is found.

If all of the action items are checked, and the request is still not fully handled, *DispatchAction* calls the *OnAction* event handler of the default action item.

Finally, if the request has been handled by the *BeforeDispatch* event handler or any of the action items, *DispatchAction* generates an *AfterDispatch* event. The *AfterDispatch* event does not occur if the *BeforeDispatch* event or any of the action items has already sent the response.

DispatchAction returns a *Boolean* value indicating whether the Web application should send the response message if it is not already sent. *DispatchAction* returns *True* if the action was handled by the action items and the *AfterDispatch* event handler did not change its *Handled* parameter to *False*.

See also *Actions* property, *AfterDispatch* event, *BeforeDispatch* event, *TWebActionItem* object, *TWebActionItem.OnAction* event, *TWebApplication.HandleRequest* method, *TWebRequest* object, *TWebResponse* object

DoAfterDispatch

DoAfterDispatch generates an *AfterDispatch* event and returns a value that indicates whether the Web application can send the response.

```
function DoAfterDispatch(Request: TWebRequest; Response: TWebResponse): Boolean;
```

Description Applications cannot call *DoAfterDispatch*. The *DispatchAction* method calls *DoAfterDispatch* after sending the HTTP request message to any matching action items. The *DispatchAction* method does not call *DoAfterDispatch* if the HTTP request message was not handled by the action items or by a *BeforeDispatch* event handler. *DoAfterDispatch* is also not called if the message was already sent back to the Web client.

If the *AfterDispatch* event handler indicates that the response should not be sent, *DoAfterDispatch* returns *False*, and the Web application drops the connection without sending the response message. Otherwise, *DoAfterDispatch* returns *True*, and the Web application sends the response message if it was not sent from within the *AfterDispatch* event handler.

See also *AfterDispatch* event, *BeforeDispatch* event, *DispatchAction* method

DoBeforeDispatch

DoBeforeDispatch generates a *BeforeDispatch* event and returns a value that indicates whether the dispatcher should look for action items to handle the request.

```
function DoBeforeDispatch(Request: TWebRequest; Response: TWebResponse): Boolean;
```

Description Applications cannot call *DoBeforeDispatch*. The *DispatchAction* method calls *DoBeforeDispatch* before trying to match the HTTP request message with any of its action items.

If the *BeforeDispatch* event handler indicates that it has handled the request message, *DoBeforeDispatch* returns *True*, and the *DispatchAction* method does not try to pass the request message to any of the action items. Otherwise, *DoBeforeDispatch* returns *False*, and unless the response message was sent from within the *BeforeDispatch* event handler, the *DispatchAction* method proceeds with looking for action items to handle the request message.

If *DoBeforeDispatch* returns *True* but the response has not been sent, *DispatchAction* generates an *AfterDispatch* event.

See also *AfterDispatch* event, *BeforeDispatch* event, *DispatchAction* method

Notification

Notification overrides the inherited method to introduce the dispatcher to *TCustomContentProducer* objects when they are added to the application.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description *Notification* ensures that all *TCustomContentProducer* objects have access to the dispatcher, by setting their *Dispatcher* property as they are added to the application.

See also *TCustomContentProducer.Dispatcher* property

TDatabase

Unit: DBTables — Palette: Data Access

TObject • TPersistent • TComponent • TDatabase

TDatabase provides discrete control over a connection to a single database in a database application.

Description Use *TDatabase* when a database application requires any of the following control over a database connection:

- Persistent database connections
- Transaction control
- Customized database server logins
- Application-specific BDE aliases

TDatabase is especially important for the control it permits over database transaction processing with the BDE when connected to a remote SQL database server.

Note Explicit declaration of a *TDatabase* component for each database connection in an application is optional if the application does not require any of these controls for that connection. If a *TDatabase* component is not explicitly

declared and instantiated for a database connection, a temporary database component with a default set of properties is created for it at runtime.

See also *TSession* object, *TSessionList* object

Properties

The following table lists the properties of *TDatabase*:

Property	Visibility	Default	Documented in
AliasName	published		TDatabase
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Connected	published	False	TDatabase
DatabaseName	published		TDatabase
DataSetCount	public, read-only		TDatabase
DataSets	public, read-only		TDatabase
DesignInfo	public		TComponent
Directory	public		TDatabase
DriverName	published		TDatabase
Handle	public		TDatabase
InTransaction	public, read-only		TDatabase
IsSQLBased	public, read-only		TDatabase
KeepConnection	published	True	TDatabase
Locale	public, read-only		TDatabase
LoginPrompt	published	True	TDatabase
Name	published		TComponent
Owner	public, read-only		TComponent
Params	published		TDatabase
Session	public, read-only		TDatabase
SessionAlias	public, read-only		TDatabase
SessionName	published		TDatabase
Tag	published	0	TComponent
Temporary	public		TDatabase
TraceFlags	public		TDatabase
TransIsolation	published	tiReadCommitted	TDatabase
VCLComObject	public		TComponent

D

AliasName

Specifies the Borland Database Engine (BDE) alias used by this database connection.

property AliasName: **string**;

Description Use *AliasName* to specify the name of an existing BDE alias for the database component. A BDE alias contains database configuration information for connection to a specific kind of database, such as InterBase, dBASE, or Paradox. The configuration information associated with a BDE alias differs by database type.

If an application sets *AliasName*, any value previously assigned to the *DriverName* property is cleared because the alias automatically specifies a driver name as one of its parameters. Other alias connection information is stored in the *Params* property when *AliasName* is set.

Note Attempting to set *AliasName* when the *Connected* property is *True* raises an exception.

Note An application can specify the *DatabaseName*, *DriverName*, and *Params* properties rather than setting *AliasName* if the desired BDE alias configuration does not already exist. An application can also provide an application-specific "alias" name by setting *DatabaseName* when *AliasName* is already set.

Tip At design time double-click a *TDatabase* component to invoke the Database editor and set the *AliasName*.

See also *Connected* property, *DatabaseName* property, *DriverName* property, *Params* property

Connected

Indicates whether or not a database connection is active.

property Connected: Boolean;

Description Set *Connected* to *True* to establish a database connection without opening a dataset. Set *Connected* to *False* to close a database connection. An application can check *Connected* to determine the current status of a database connection. If *Connected* is *True*, the database connection is active; if *False*, and the *KeepConnection* property is also *False*, then the connection is inactive.

Note Set *KeepConnection* to *True* to avoid having to log in to the server each time a database connection is reopened.

See also *DataSets* property, *KeepConnection* property

DatabaseName

Specifies the name of the database to associate with this database component.

property DatabaseName: **string**;

Description Use *DatabaseName* to specify the name of the database to use with a database component. If *DatabaseName* is the same as an existing Borland Database Engine (BDE) alias, then the *AliasName* and *DriverName* properties need not be set. If *DatabaseName* does not match an existing BDE alias, then either the application must also supply a valid alias in the *AliasName* property in addition to the *DatabaseName*, or it must supply the *DriverName* and *Params* properties.

DatabaseName can be a fully qualified path name when connecting to a Paradox or dBASE database. If a full path name is provided, the application must supply the *DriverName* and *Params* properties.

Note Attempting to set *DatabaseName* when the *Connected* property is *True* raises an exception.

Tip At design time double-click a *TDatabase* component to invoke the Database editor and set the *DatabaseName*.

See also *AliasName* property, *Connected* property, *DriverName* property, *Params* property

DataSetCount

Indicates the number of active datasets associated with the database component.

property DataSetCount: Integer;

Description Use *DataSetCount* to determine the number of currently active datasets associated with the database component. As datasets are opened and closed this value changes appropriately.

For example, the following code fragment illustrates how *DataSetCount* can be used to ensure that an action is taken for every open dataset that is a table:

```
var
  I: Integer;
begin
  for I := 0 to DataSetCount - 1 do
    if DataSets[I] is TTable then
      DataSets[I].CachedUpdates := True;
end;
```

See also *DataSets* property, *Open* method, *Close* method

D

DataSets

Provides an indexed array of all active datasets for a database component.

property DataSets[Index: Integer]: TDBDataSet;

Description Use *DataSets* to access active datasets associated with a database component. An active dataset is one that is currently open. For example, the following code fragment illustrates how *DataSets* can be used with the *DataSetCount* property to ensure that an action is taken for every open dataset that is a table:

```
var
  I: Integer;
begin
  for I := 0 to DataSetCount - 1 do
    if DataSets[I] is TTable then
      DataSets[I].CachedUpdates := True;
end;
```

See also *DataSetCount* property, *TDBDataSet* object

Directory

Specifies the working directory for a Paradox or dBASE database.

property Directory: **string**

Description Use *Directory* at runtime to change or retrieve the working directory for a Paradox or dBASE database. This is the location these databases use to create temporary tables at runtime.

Ordinarily a directory location is specified in the PATH parameter of the database alias (specified by the *AliasName* property). At runtime, however, some applications may need to change the working directory after a database is opened.

To change the value of *Directory* at runtime, a database must already be open, or an exception is raised. Clearing the current value of *Directory* sets its value to the original working directory specified when the database was first opened.

Note Do not use *Directory* when connected to remote databases. An exception is raised in these circumstances. An exception is also raised if an application accesses *Directory* when the database is closed.

See also *AliasName* property, *DatabaseName* property, *Open* method

DriverName

Specifies the name of the BDE driver for the database.

property DriverName: **string**;

Description Use *DriverName* to specify the name of the BDE driver to use for databases connections that do not specify an *AliasName*, but that do specify a *DatabaseName*. *DriverName* must be a valid BDE driver (for example, STANDARD, ORACLE, SYBASE, or INTERBASE).

Note If an application sets *DriverName*, it must also specify connection parameters in the *Params* property. Ordinarily the alias specified in the *AliasName* property supplies connection parameters, but when *DriverName* is set, the *AliasName* property is automatically cleared to avoid BDE driver name contention.

Attempting to set *DriverName* when the *Connected* property is *True* raises an exception.

Tip At design time double-click a *TDatabase* component to invoke the Database editor and set the *DriverName*.

See also *AliasName* property, *Connected* property, *DatabaseName* property, *Params* property

Handle

Specifies the Borland Database Engine (BDE) database handle.

```
type
  HDBIDB: Longint;

property Handle: HDBIDB;
```

Description Use *Handle* only to bypass *TDatabase* methods and make direct calls to the directly to the BDE API. Many BDE function calls require a handle parameter. *Handle* is assigned an initial value when a database is opened.

Note Do not use this property unless an application requires BDE functionality not available through standard Delphi components.

See also *IsSQLBased* property, *Locale* property

InTransaction

Indicates whether a database transaction is in progress or not.

```
property InTransaction: Boolean
```

Description Examine *InTransaction* at runtime to determine if a database transaction is currently in progress. *InTransaction* is *True* if a transaction is in progress, *False* otherwise.

The value of *InTransaction* cannot be changed directly. Calling *StartTransaction* sets *InTransaction* to *True*. Calling *Commit* or *Rollback* sets *InTransaction* to *False*.

See also *Commit* method, *Rollback* method, *StartTransaction* method

D

IsSQLBased

Indicates if a database component is using either the Borland Database Engine (BDE) SQL Links driver or the BDE ODBC socket.

```
property IsSQLBased: Boolean;
```

Description Examine *IsSQLBased* at runtime to determine if a database connection is using a BDE SQL Links driver, the ODBC socket, or the Paradox, dBASE, and ASCII text file STANDARD driver. If *IsSQLBased* is *True*, the connection is using an SQL Links driver or is using the ODBC socket to communicate with a third-party ODBC driver. If the connection uses the STANDARD driver, *IsSQLBased* is *False*.

See also *Handle* property, *Locale* property

KeepConnection

Specifies whether an application remains connected to a database even if no datasets are open.

```
property KeepConnection: Boolean;
```

Description Use *KeepConnection* to specify whether an application remains connected to a database even if no datasets are currently open. When *KeepConnection* is *True* (the default) the connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, set *KeepConnection* to *True* to reduce network traffic, speed up applications, and avoid logging in to the server each time the connection is reestablished.

When *KeepConnection* is *False* a connection is dropped when there are no open datasets. Dropping a connection releases system resources allocated to the connection, but if a dataset is later opened that uses the database, the connection must be reestablished and initialized.

Note The *KeepConnection* setting for temporary database components created automatically by Delphi as needed is determined by the *KeepConnections* property of *TSession*.

See also *Close* method, *Open* method, *TSession.KeepConnections* property

Locale

Identifies the Borland Database Engine (BDE) language driver for the database component.

```
type
  TLocale: Pointer;

property Locale: TLocale;
```

Description Examine *Locale* to determine the BDE language driver used by the database component. Applications that make direct calls to the BDE may also need to pass *Locale* information as an API function parameter.

Language driver information may also be available as one of the values in the Params property when a database component is using a BDE alias.

Note Do not use this property unless an application requires BDE functionality not available through standard Delphi components.

See also *AliasName* property, *Handle* property, *IsSQLBased* property, *Params* property

LoginPrompt

Indicates whether or not to display a standard Login dialog box when establishing a database connection.

```
property LoginPrompt: Boolean;
```

Description Use *LoginPrompt* to control the login method for remote database connections. If *True*, (the default), the standard Delphi Login dialog box opens when the application attempts to connect to a database. The standard Login dialog box prompts for a valid user name and password. If the entries provided by the user are not valid, the connection fails.

If an application sets *LoginPrompt* to *False*, then user name and password values must be supplied either in an *OnLogin* event, or as USER NAME and PASSWORD parameters in the *Params* property.

Note Storing hard-coded user name and password entries in the *Params* property or in code for an *OnLogin* event can compromise server security.

See also *Params* property, *OnLogin* event

Params

Contains database connection parameters for the Borland Database Engine (BDE) alias associated with the database component.

```
property Params: TStrings;
```

Description Use *Params* to examine or specify database connection parameters, such as path name, server name, schema-caching size, language driver, user name, and password. Actual parameters in *Params* differ depending on the current BDE alias and driver for the database component.

Params is a list of string items, each representing a different database connection parameter. If the *AliasName* property specifies a valid BDE alias, then *Params* automatically contains the parameters defined for that alias. If, instead of providing an alias, an application uses the *DriverName* property to provide a local alias, the application must provide *Params* values directly.

Tip At design time double-click a *TDatabase* component to invoke the Database editor and set *Params*.

See also *AliasName* property, *DatabaseName* property, *LoginPrompt* property, *OnLogin* event

Session

Points to the session component with which this database component is associated.

property Session: TSession

Description Use *Session* to determine the session component that controls the database component. By default, a database component is associated with the default session component, *Session*, that is automatically created for all database applications. To assign a database component to a different session in a multi-threaded application, specify the name of a different session component in the *SessionName* property.

See also *SessionAlias* property, *SessionName* property

SessionAlias

Specifies whether or not a database component is using a session alias.

property SessionAlias: Boolean;

Description Examine *SessionAlias* to see if a database component is using a session alias. If *SessionAlias* is *True*, a session alias is in use. If *False*, a session alias is not in use.

A session alias is automatically created for a database component if it uses the *DriverName* and *Params* properties instead of the *AliasName* property, and the *Params* property contains any parameters. A session alias may also be created if an application permits the user to log into a server using a different user name than the one specified in the *Params* property.

A session alias exists only while the application is running. It is not stored in the BDE configuration file, and cannot be accessed from outside the application.

See also *AliasName* property, *DriverName* property, *IsSQLBased* property, *Params* property, *Session* property, *SessionName* property

SessionName

Identifies the name of the session used by this database component.

property SessionName: **string**;

Description Use *SessionName* to specify the session with which a database component is associated. If *SessionName* is blank, a database component is automatically associated with the default session, *Session*.

To associate a database component with a different session in a database application *SessionName* must matches the *SessionName* property of an existing session component.

See also *Session* property, *SessionAlias* property, *TSession* object, *TSessionList* object, *TSession.SessionName* property

Temporary

Indicates whether a database component is a temporary one, created by the system as needed, or a persistent one explicitly created, managed, and freed within the application.

```
property Temporary: Boolean;
```

Description Examine *Temporary* to determine if a database component is a temporary one, created by the system as needed, or a persistent one explicitly created, managed, and freed within the application. *Temporary* is *True* if the system created the component, *False* otherwise.

A temporary database component is created when a dataset is opened and the dataset is not already associated with an existing database component. If *Temporary* remains *True*, then a temporary database component is freed when the dataset is closed. An application can prevent the destruction of a temporary database component by setting *Temporary* to *False* while the dataset is active, but the application is then responsible for closing the database when it is no longer needed.

See also *Create* method, *Free* method

TraceFlags

Specifies the database operations to track with the SQL Monitor at runtime.

```
type
  TTraceFlag = (tfQPrepare, tfQExecute, tfError, tfStmt, tfConnect, tfTransact,
  tfBlob, tfMisc, tfVendor);

  TTraceFlags = set of TTraceFlag;

property TraceFlags: TTraceFlags
```

Description Use *TraceFlags* to specify which database operations the SQL Monitor should track in an application at runtime. *TraceFlags* is only meaningful for the SQL Monitor, which is provided to enable performance tuning when working with remote SQL database servers.

Note Normally trace options are set from the SQL Monitor rather than setting *TraceFlags* in application code.

The initial value of a database component's *TraceFlags* property is determined by the *TraceFlags* property of the session component that owns it.

The *TTraceFlags* type defines the individual values that can be included in the *TraceFlags* property. The following table summarizes those values:

Values	Meaning
tfQPrepare	Monitor Prepare statements.
tfQExecute	Monitor ExecSQL statements.
tfError	Monitor server error messages. Such messages may include an error code.
tfStmt	Monitor all SQL statements.
tfConnect	Monitor database connect and disconnect operations, including allocation of connection handles, and freeing connection handles.
tfTransact	Monitor transaction statements, such as StartTransaction, Commit, and Rollback.
tfBlob	Monitor operations on blob data types.
tfMisc	Monitor any statements not covered by other flag options.
tfVendor	Monitor direct API function calls to the database server.

Because *TraceFlags* is a set property, an application can specify different combinations of flags to monitor different combinations of statements. For example, the following statement limits monitoring to database connections and SQL statement preparation:

```
TraceFlags := [tfConnect, tfQPrepare];
```

See also *Session* property, *SessionName* property, *TSession.TraceFlags* property

TransIsolation

Specifies the transaction isolation level for transactions managed by the Borland Database Engine (BDE).

```
type
  TTransIsolation = (tiDirtyRead, tiReadCommitted, tiRepeatableRead);

property TransIsolation: TTransIsolation;
```

Description Use *TransIsolation* to specify the transaction isolation level for database transactions managed by the BDE. Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables, and how much a transaction sees of the work performed by other transactions.

Note Applications that use passthrough SQL for handling transactions must pass a transaction isolation level directly to the database server using the appropriate SQL statement.

TransIsolation can be any one of the three values summarized in the following table:

Isolation level	Meaning
tiDirtyRead	Permits reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level a transaction is least isolated from the effects of other transactions.
tiReadCommitted	Permits reading of committed (permanent) changes made to the database by other simultaneous transactions. This is the default TransIsolation property value.
tiRepeatableRead	Permits a single, one-time reading of the database. The transaction cannot see any subsequent changes made by other simultaneous transactions. This isolation level guarantees that once a transaction reads a record, its view of that record does not change unless it makes a modification to the record itself. At this level, a transaction is most isolated from other transactions.

Different database servers support different levels of transaction isolation. If an application sets *TransIsolation* to an unsupported level for a remote SQL server, the BDE uses the next highest level supported by that server. The following table summarizes the isolation levels supported by the servers recognized by the BDE:

Server	Specified Level	Actual Level
Oracle	tiDirtyRead	tiReadCommitted
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead (READONLY)
Sybase, MS-SQL	tiDirtyRead	tiReadCommitted
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	Not supported
DB2	tiDirtyRead	tiDirtyRead
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
Informix	tiDirtyRead	tiDirtyRead
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
InterBase	tiDirtyRead	tiReadCommitted
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
Paradox, dBASE	tiDirtyRead	tiDirtyRead
	tiReadCommitted	Not supported
	tiRepeatableRead	Not supported

Note For local transactions against Paradox and dBASE, *TransIsolation* must be set to *tiDirtyRead*. Otherwise, an exception is raised.

If an application uses ODBC to interface with a server, the ODBC driver must also support the isolation level. For more information about supported isolation levels, see the documentation for the ODBC driver in question.

See also *AliasName* property, *DriverName* property, *Params* property

Events

The following table lists the event for *TDatabase*:

Event	Visibility	Documented in
OnLogin	published	TDatabase

OnLogin

OnLogin occurs when an application connects to a database.

```
TLoginEvent = procedure(Database: TDatabase; LoginParams: TStrings) of object;
property OnLogin: TLoginEvent;
```

Description Write an *OnLogin* event handler to take specific actions when an application attempts to connect to a database. By default, when an *OnLogin* event occurs, the current USER NAME is read from the *Params* property, and a standard Login dialog box opens. The dialog prompts for a user name and password combination, and then uses the values entered by the user to set the USER NAME and PASSWORD values in the *Params* property. These values are then passed to the remote server.

Applications that provide alternative *OnLogin* event handlers must set the USER NAME and PASSWORD values in *LoginParams*, but once a connection is established, the PASSWORD value can be discarded to avoid compromising server security.

See also *Connected* property, *KeepConnection* property, *Params* property, *Open* method

Methods

The following table lists the methods of *TDatabase*:

Method	Visibility	Documented in
ApplyUpdates	public	TDatabase
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Close	public	TDatabase
CloseDataSets	public	TDatabase

Method	Visibility	Documented in
Commit	public	TDatabase
Create	public	TDatabase
DefaultHandler	public	TObject
Destroy	public	TDatabase
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FlushSchemaCache	public	TDatabase
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Open	public	TDatabase
RemoveComponent	public	TComponent
Rollback	public	TDatabase
SafeCallException	public	TComponent
StartTransaction	public	TDatabase
ValidateName	public	TDatabase

ApplyUpdates

Posts pending cached updates for specified datasets to the database server.

```
procedure ApplyUpdates(const DataSets: array of TDataSet);
```

Description Call *ApplyUpdates* to post pending cached updates for a specific set of open datasets to the database server. *ApplyUpdates* is only meaningful if the *CachedUpdates* property of a specified dataset is *True*.

D

DataSets is a list of dataset names specifying the datasets for which to post pending updates. *DataSets* need not list every currently open dataset. For each listed dataset *ApplyUpdates* calls the dataset's *ApplyUpdates* and *CommitUpdates* methods to post that dataset's pending cached updates.

Applying updates is a two-phase process that takes place within the context of the database component's transaction control. When an application calls *ApplyUpdates*, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (Phase 1).

   If the database write is successful:

   1. Database changes are committed, ending the transaction.
   2. Cached updates are committed, clearing the internal cache buffer (Phase 2).

   If the database write fails database changes are rolled back, ending the transaction.

The two-phased approach allows for effective error recovery, especially when updating multiple and interrelated datasets (for example, the datasets associated with a master/detail form).

See also *Commit* method, *Open* method, *Rollback* method, *StartTransaction* method, *TDataSet.CachedUpdates* property, *TDataSet.ApplyUpdates* method, *TDataSet.CommitUpdates* method

Close

Closes all open datasets associated with the database component and disconnects from the database server.

```
procedure Close;
```

Description Call *Close* to disconnect from a database server. Disconnecting frees system resources allocated to the connection.

Note *Close* is called automatically when there are no more open datasets associated with the database component and the *KeepConnection* property is *False*. A database connection can also be closed by setting the *Connected* property to *False*.

See also *Connected* property, *KeepConnection* property, *CloseDataSets* method

CloseDataSets

Closes all datasets associated with the database component without disconnecting from the database server.

```
procedure CloseDatasets;
```

Description Call *CloseDataSets* to close all active datasets without disconnecting from the database server. Ordinarily, when an application calls *Close*, all datasets are closed, and the connection to the database server is dropped. Calling *CloseDataSets* instead of *Close* ensures that an application can close all active datasets without having to reconnect to the database server at a later time.

See also *KeepConnection* property, *Close* method

Commit

Permanently stores updates, insertions, and deletions of data associated with the current transaction, and ends the current transactions.

```
procedure Commit;
```

Description Call *Commit* to permanently store to the database server all updates, insertions, and deletions of data associated with the current transaction and then end the transaction. The current transaction is the last transaction started by calling *StartTransaction*.

Note Before calling *Commit*, an application may check the status of the *InTransaction* property. If an application calls *Commit* and there is no current transaction, an exception is raised.

See also *InTransaction* property, *Rollback* method, *StartTransaction* method

Create

Creates an instance of a *TDatabase* component.

```
constructor Create(AOwner: TComponent);
```

Description Call *Create* to instantiate a database component at runtime. An application can create a database component in order to control the component's existence and set its properties and events, or an application can let Delphi create temporary database components as needed at runtime.

Create instantiates a database component and:

- Sets the *Session* property to the active session.
- Sets the *SessionName* property to the name of the active session component.
- Adds this component to the list of database components for the session.
- Creates an empty list of dataset components for the *DataSets* property.
- Creates an empty string list for the *Params* property.
- Sets the *LoginPrompt* property and the *KeepConnection* property to *True*.
- Sets the *Locale* property to the locale of the session.
- Sets the *TransIsolation* property to *tiReadCommitted*.

See also *DataSets* property, *KeepConnection* property, *Locale* property, *LoginPrompt* property, *Params* property, *Session* property, *SessionName* property, *TransIsolation* property, *Destroy* method

Destroy

Destroys the instance of a database component.

```
destructor Destroy;
```

Description Do not call *Destroy* directly in an application. Instead, an application should call *Free*, which verifies that the database component is not already freed before calling *Destroy*.

Destroy closes all active datasets and disconnects from the database server, if necessary. It then frees the string resources allocated for the *Params* and *DataSets* properties before calling its inherited destructor.

See also *DataSets* property, *Params* property, *Create* method, *Free* method

FlushSchemaCache

Flushes the cached schema information for a table.

```
procedure FlushSchemaCache(const TableName: string);
```

Description Call *FlushSchemaCache* if the schema information for an SQL table is changed (for example, because a field, index, or table is added to or dropped from a database).

When an application opens an SQL table, schema information that describes the field names, field types, and index information is stored in memory. If an application changes the schema for an SQL table, it should call *FlushSchemaCache* to update the schema information in memory.

See also *Open* method

Open

Connects to a database server.

```
procedure Open;
```

Description Call *Open* to establish a connection to a local or remote database server. *Open* verifies that the database specified by the *DatabaseName* or *Directory* properties exists, and that the user name and password are valid. If so, it connects to the database. If an *OnLogin* event exists for the database component, it is executed. Otherwise, the default Login dialog box appears.

Note Setting *Connected* to *True* also connects to the database server.

See also *Connected* property, *DatabaseName* property, *Directory* property, *OnLogin* event

Rollback

Cancels all updates, insertions, and deletions for the current transaction and ends the transaction.

```
procedure Rollback;
```

Description Call *Rollback* to cancel all updates, insertions, and deletions for the current transaction and to end the transaction. The current transaction is the last transaction started by calling *StartTransaction*.

Note Before calling *Rollback*, an application may check the status of the *InTransaction* property. If an application calls *Rollback* and there is no current transaction, an exception is raised.

See also *InTransaction* property, *Commit* method, *StartTransaction* method

StartTransaction

Begins a new transaction against the database server.

```
procedure StartTransaction;
```

Description Call *StartTransaction* to begin a new transaction against the database server. Before calling *StartTransaction*, an application should check the status of the *InTransaction* property. If *InTransaction* is *True*, indicating that a transaction is already in progress, a subsequent call to *StartTransaction* without first calling *Commit* or *Rollback* to end the current transaction raises an exception.

Updates, insertions, and deletions that take place after a call to StartTransaction are held by the server until an application calls Commit to save the changes or Rollback is to cancel them.

See also *InTransaction* property, *Commit* method, *Rollback* method

ValidateName

Raises an exception if a specified database is already open in the active session.

```
procedure ValidateName(const Name: string);
```

Description Call *ValidateName* to prevent duplicate access to a database from within a single session. *Name* is a string containing the name of the database to test. If the database is already open, *ValidateName* raises the *EDatabaseError* exception. If the database is not open, the procedure returns, and the application continues processing.

Note Most applications should not need to call this method directly. It is called automatically each time a database is opened.

See also *DatabaseName* property, *Open* method, *EDatabaseError* object

TDataModule

Unit: Forms

TObject • TPersistent • TComponent • TDataModule

TDataModule is a specialized class for centralized handling of nonvisual components in an application.

Description

Use a *TDataModule* object in an application to provide a location for centralized handling of nonvisual components in an application. Typically these are data access components (*TSession*, *TSessionList*, *TDatabase*, *TTable*, *TQuery*, *TStoredProc*, and *TBatchMove*), but they can also be other nonvisual components, such as *TTimer*, *TOpenDialog*, *TImageList*, or *TOleContainer*).

At design time a *TDataModule* object provides a visual container into which a developer can place nonvisual components, set their properties, and write event handlers for them. To create a data module at design time, choose File | New Data Module.

In the unit file for the data module a developer may also place any business rules that are to be applied to the application.

To make the data module available to another unit in the application, select that unit, then choose File | Use Unit to add the data module to the **uses** clause for the unit.

See also

TForm object

Properties

The following table lists the properties of *TDataModule*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
DesignOffset	public		TDataModule
DesignSize	public		TDataModule
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
VCLComObject	public		TComponent

DesignOffset

Specifies the design offset for the data module at design time.

```
property DesignOffset: TPoint;
```

Description Specifies the design offset for the data module at design time. An application should never need to set this value.

See also *DesignSize* property

DesignSize

Specifies the design size for the data module at design time.

```
property DesignSize: TPoint;
```

Description Specifies the design size for the data module at design time. An application should never need to set this value.

See also *DesignOffset* property

Events

The following table lists the events of *TDataModule*:

Event	Visibility	Documented in
OnCreate	published	TDataModule
OnDestroy	published	TDataModule

OnCreate

OnCreate occurs when an application instantiate a data module.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property OnCreate: TNotifyEvent;
```

Description Write an *OnCreate* event handler to take specific actions when an application instantiates a data module. For example, if a data module contains database and dataset components, an application may establish a database connection immediately. If the data module contains timers, the application may initialize them.

See also *OnDestroy* event

OnDestroy

Specifies the design offset for the data module at design time.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property OnDestroy: TNotifyEvent;
```

Description Write an *OnDestroy* event handler to take specific actions when an application frees a data module. For example, if the unit code for data module instantiates any objects of its own, such as string lists, the *OnDestroy* event handler can be used to free those objects.

See also *OnCreate* event

Methods

The following table lists the methods of *TDataModule*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TDataModule
CreateNew	public	TDataModule
DefaultHandler	public	TObject
Destroy	public	TDataModule
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject

Method	Visibility	Documented in
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Create

Creates an instance of a data module.

constructor Create(AOwner: TComponent);

Description Call *Create* to instantiate a data module at runtime if it was not created at design time. *Create* calls *CreateNew*. If an error occurs, an exception is raised. Otherwise Create calls the OnCreate event handler for the data module if one is assigned to it.

See also *CreateNew* method, *Destroy* method, *OnCreate* event

CreateNew

Creates an instance of a data module.

constructor CreateNew(AOwner: TComponent);

Description Do not call *CreateNew* directly. Instead, call *Create*. *CreateNew* calls its inherited *Create* constructor, and then calls the *TScreen.AddDataModule* method.

See also *Create* method, *Destroy* method, *TScreen.AddDataModule* method

Destroy

Destroys the instance of the data module.

destructor Destroy;

Description Do not call *Destroy* directly. Instead call *Free* to verify that the query is not already freed before calling *Destroy*. *Destroy* removes any fixup references, calls its *OnDestroy* event handler if one is defined for it, calls *TScreen.RemoveDataModule*, and then calls its inherited *Destroy* destructor.

See also *Create* method, *TScreen.RemoveDataModule* method

D

TDataSet

Unit: DB

TObject • TPersistent • TComponent • TDataSet

TDataSet is the virtualized base class for all dataset components that represent data in rows and columns in a database.

Description

TDataSet is a virtualized base class that encapsulates a database-engine independent set of properties, events, and methods for working with data. Many of the properties, events, and methods in *TDataSet* are **abstract** or **virtual**.

Abstract declarations are just that: declarations without implementations, meaning that at the *TDataSet* level they cannot be used or accessed. Instead developers must use or derive descendents of *TDataSet* that declare and implement these **abstract** methods. Virtual methods are declared and implemented in *TDataSet*, but in many cases, these methods are merely stubs that are reimplemented in descendents such as *TClientDataSet*, *TBDEDataSet*, *TDBDataSet*, *TQuery*, *TStoredProc*, and *TTable*.

For example, both single-tiered and multi-tiered database application that uses distributed datasets accesses data using *TClientDataSet*, a direct descendent of *TDataSet*. Developers who develop their own distributed dataset objects should derive them from *TClientDataSet* as well.

Similarly, a traditional Delphi database application accesses data through the Borland Database Engine (BDE) using *TQuery*, *TStoredProc*, or *TTable* objects. Developers who create custom dataset components that use the BDE directly in a client application derive them from *TBDEDataSet*, *TDBDataSet*, *TQuery*, *TStoredProc*, or *TTable*.

See also

TClientDataSet object, *TBDEDataSet* object, *TDBDataSet* object, *TQuery* object, *TStoredProc* object, *TTable* object

Properties

The following table lists the properties of *TDataSet*:

Property	Visibility	Default	Documented in
Active	public	False	TDataSet
AutoCalcFields	public	True	TDataSet
BOF	public, read-only		TDataSet
Bookmark	public		TDataSet
CanModify	public, read-only		TDataSet
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent

Property	Visibility	Default	Documented in
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DataSource	public, read-only		TDataSet
DefaultFields	public, read-only		TDataSet
Designer	public, read-only		TDataSet
DesignInfo	public		TComponent
EOF	public, read-only		TDataSet
FieldCount	public, read-only		TDataSet
FieldDefs	public		TDataSet
Fields	public		TDataSet
FieldValues	public		TDataSet
Filter	public		TDataSet
Filtered	public	False	TDataSet
FilterOptions	public	[]	TDataSet
Found	public, read-only		TDataSet
Modified	public, read-only		TDataSet
Name	published		TDataSet
Owner	public, read-only		TComponent
RecNo	public		TDataSet
RecordCount	public, read-only		TDataSet
RecordSize	public, read-only		TDataSet
State	public, read-only		TDataSet
Tag	published	0	TComponent
VCLComObject	public		TComponent

Active

Specifies whether or not a dataset is open.

property Active: Boolean;

Description

Use *Active* to determine or set a dataset's connection to data in a database. When *Active* is *False*, the dataset is closed; the dataset cannot read data from or write data to the database. When *Active* is *True*, data can be read from and written to the database.

Setting *Active* to *True*:

- Triggers the *BeforeOpen* event handler if one is defined for the dataset.
- Sets the dataset state to *dsBrowse*.
- Opens a cursor into the dataset.
- Triggers the *After* Open event handler if one is defined for the dataset.

If an error occurs during the dataset open, dataset state is set to *dsInactive*, and the cursor is closed.

D

An application must set *Active* to *False* before changing other properties that affect the status of the database or the controls that display data in an application.

Note Calling the *Open* method sets *Active* to *True*; calling the *Close* method sets *Active* to *False*.

See also *DataSource* property, *State* property, *AfterOpen* event, *BeforeOpen* event, *Close* method, *Open* method

AutoCalcFields

Determines when the *OnCalcFields* event is triggered.

property AutoCalcFields: Boolean;

Description Set *AutoCalcFields* to control when the *OnCalcFields* event is triggered to update calculated fields during dataset processing. A calculated field is one that derives its value from the values of one or more fields in the dataset, sometime with additional processing. When *AutoCalcFields* is *True* (the default), *OnCalcFields* is triggered when:

- A dataset is opened.
- A dataset is put into *dsEdit* state.
- Focus moves from one visual control to another, or from one column to another is a data-aware grid control and modifications were made to the record.
- A record is retrieved from a database.

To reduce the frequency with which *OnCalcFields* is called, set *AutoCalcFields* to *False*. When *AutoCalcFields* is *False*, *OnCalcFields* is not called when changes are made to individual fields within a record. If an application permits users to change data, *OnCalcFields* is frequently triggered. In these cases an application may set *AutoCalcFields* to *False* to reduce the frequency with which *AutoCalcFields* is called.

See also *OnCalcFields* event

BOF

Indicates whether or not a cursor is positioned at the first record in a dataset.

property BOF: Boolean;

Description Test *BOF* (beginning of file) to determine if the cursor is positioned at the first record in a dataset. If *BOF* is *True*, the cursor is unequivocally on the first row in the dataset. *BOF* is *True* when an application:

- Opens a dataset.
- Calls a dataset's *First* method.

- Call a dataset's *Prior* method, and the method fails (because the cursor is already on the first row in the dataset).
- Calls *SetRange* on an empty range or dataset.

BOF is *False* in all other cases. An application should assume *BOF* is *False* unless one of the conditions above is met and the application tests the property directly.

See also *EOF* property, *First* method, *Prior* method, *TTable.SetRange* method

Bookmark

Specifies the current bookmark in the dataset.

```
type
  TBookmarkStr: string;

property Bookmark: TBookmarkStr;
```

Description *Bookmark* gets or sets the current bookmark in a dataset. A bookmark provides a convenient way to mark a location in a dataset so that an application can easily return to that location quickly.

An application can read *Bookmark* to retrieve the bookmark associated with the current record, and it can position the cursor in the dataset by assigning a saved bookmark value to this property.

See also *GetBookmark* method, *GotoBookmark* method, *FreeBookmark* method

CanModify

Indicates whether the database underlying a dataset permits write access to data.

```
property CanModify: Boolean;
```

Description This property is always *True* for an instance of a *TDataSet* object. Descendent objects, such as *TQuery*, *TStoredProc*, and *TTable*, redeclare and reimplement this property to perform meaningful tests to determine whether a dataset has write access to data.

See also *DataSource* property, *TDataSource.AutoEdit* property

DataSource

Returns **nil** for datasets that cannot specify a data source.

```
property DataSource: TDataSource;
```

Description *DataSource* is a property that returns **nil** if a dataset cannot specify its own data source.

Descendant dataset objects that support specifying a data source redeclare and implement methods for getting and setting the *DataSource* property.

See also *Active* property, *IsLinkedTo* method, *TDataSource* object

DefaultFields

Indicates whether a dataset's underlying field components are generated dynamically when the dataset is opened.

property DefaultFields Boolean;

Description Read *DefaultFields* to determine whether or not a dataset uses dynamically generated field components, or persistent field components. If *DefaultFields* is *True*, the dataset uses dynamically allocated field components. If *DefaultFields* is *False*, the dataset uses persistent field components.

A dataset always creates dynamic field components based on the structure of its underlying database table or tables when the dataset is opened unless persistent field components are assigned to a dataset at design time using the Fields editor.

See also *Active* property, *Fields* property, *Open* method

Designer

Returns a **nil** pointer to the Dataset designer for the dataset.

property Designer: TDataSetDesigner;

Description *Designer* is a property that returns **nil** for a dataset that does not specify a Dataset designer. A Dataset designer, such as the Fields editor, enables a developer to specify the fields that belong to a dataset at design time.

Descendant dataset objects that support specifying a Dataset designer redeclare and implement methods for getting and setting the *Designer* property.

See also *DefaultFields* property, *Fields* property

EOF

Indicates whether or not a cursor is positioned at the last record in a dataset.

property EOF: Boolean;

Description Test *EOF* (end-of-file) to determine if the cursor is positioned at the last record in a dataset. If *EOF* is *True*, the cursor is unequivocally on the last row in the dataset. *EOF* is *True* when an application:

- Opens an empty dataset.
- Calls a dataset's *Last* method.
- Call a dataset's *Next* method, and the method fails (because the cursor is already on the last row in the dataset).
- Calls *SetRange* on an empty range or dataset.

EOF is *False* in all other cases. An application should assume *EOF* is *False* unless one of the conditions above is met and the application tests the property directly.

Tip If both *EOF* and *BOF* are *True*, a dataset or range is empty.

See also *BOF* property, *Last* method, *Next* method, *TTable.SetRange* method

FieldCount

Indicates the number of field components associated with the dataset.

```
property FieldCount: Integer;
```

Description Examine *FieldCount* to determine the number of fields associated with the dataset. For datasets with dynamically-created fields, *FieldCount* may differ each time a dataset is opened. For datasets with persistent fields, *FieldCount* should be unchanged each time a dataset is open.

See also *DefaultFields* property, *Designer* property, *Fields* property

FieldDefs

Points to the list of field definitions for the dataset.

```
property FieldDefs: TFieldDefs;
```

Description *FieldDefs* points to an internal list of field definitions for a dataset. While an application can examine *FieldDefs* to explore the field definitions for a dataset, it should not change these definitions unless creating a new table with *CreateTable*. Otherwise, *FieldDefs* is primarily intended for internal use by the other properties, events, and methods of *TDataSet* and its descendants.

To access fields and field values in a dataset, use the *Fields* and *FieldValues* properties, and the *FieldsByName* method.

See also *CreateTable* method, *Fields* property, *FieldValues* property, *ClearFields* method, *GetFieldList* method, *GetFieldNames* method, *FieldByName* method, *FindField* method, *SetFields* method, *TFieldDefs* object

Fields

Provides an indexed array of all field components for a dataset.

```
property Fields[Index: Integer]: TField;
```

Description Use *Fields* to access field components by index number. *Fields* is a zero-based array. If fields are generated dynamically at runtime, the order of field components in *Fields* corresponds directly to the order of columns in the table or tables underlying a dataset. If a dataset uses persistent fields, then the order of field components corresponds to the ordering of fields specified in the Fields editor at design time.

Accessing fields with the *Fields* property is useful for applications that:

- Iterate over some or all fields in a dataset.
- Work with underlying tables whose internal data structure is unknown at runtime.

If an application knows the data types of individual fields, then it can read or write individual field values through the *Fields* property. For example, the following statement assigns a field value to the *Text* property of an edit box:

```
Edit1.Text := CustTable[6].AsString;
```

The next statements assigns a value from an edit box to a field:

```
CustTable.Edit;
CustTable.Fields[6].AsString := Edit1.Text;
CustTable.Post;
```

Note The preferred method for retrieving and assigning field values is to persistent fields or the *FieldByName* method.

See also *FieldByName* method, *Edit* method, *Post* method, *TEdit* object

FieldValues

Provides access to the values for all fields in the current record for the dataset

```
property FieldValues[const FieldName: string]: Variant;
```

Description Use *FieldValues* to read and write values for fields in a dataset. *FieldValues* is the default property for *TDataSet*.

FieldName is the name of a field to read from or write to, and must be enclosed in single quotes.

FieldValues accepts and returns a Variant, so it can handle and convert fields of any type. Because *FieldValues* is the default property for *TDataSet*, you can omit the property name when referencing this property. For example, the following statements are syntactically identical and write the value from an edit box into an integer field:

```
Customers.FieldValues['CustNo'] := Edit1.Text;
```

```
Customers['CustNo'] := Edit1.Text;
```

The next statements reads a string value from a field into an edit box:

```
Customers.Edit;
Edit1.Text := Customers['Company'];
Customers.Post;
```

Note Because *FieldValues* always uses Variants, it may be a somewhat slower method of accessing data, than using a field's native format (for example, using a field's *AsXXX* property), especially in applications that process large amounts of data.

See also *Fields* property, *FieldByName* method, *Edit* method, *Post* method

Filter

Specifies the text of the current filter for a dataset.

property Filter: **string**;

Description Use *Filter* to specify a dataset filter. When filtering is applied to a dataset, only those records that meet a filter's conditions are available to an application. *Filter* contains the string that describes the filter condition. For example, the following filter condition displays only those records where the State field is 'CA' or 'MA':

```
State = 'CA' or State = 'MA'
```

When a field name contains spaces, you must enclose the field name in brackets. For example,

```
[Home State] = 'CA' or [Home State] = 'MA'
```

Note Applications can set *Filter* at runtime to change the filtering condition for a dataset at (for example, in response to user input).

See also *Filtered* property, *FilterOptions* property, *OnFilterRecord* event, *Locate* method, *Lookup* method

Filtered

Specifies whether or not filtering is active for a dataset.

property Filtered: Boolean;

Description Check *Filtered* to determine whether or not dataset filtering is in effect. If *Filtered* is *True*, then filtering is active. Otherwise *Filtered* is *False*. To apply filter conditions specified in the *Filter* property or the *OnFilterRecord* event handler, set *Filtered* to *True*.

Note When filtering is enabled, user edits to a record may mean that the record no longer meets a filter's test condition. The next time an attempt is made to the read the record from the dataset while the filter is in effect the record is not visible. If that happens, the next record that passes the filter condition becomes the current record.

See also *Filter* property, *FilterOptions* property, *OnFilterRecord* event, *Locate* method, *Lookup* method

FilterOptions

Specifies whether or not filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

```
type
  TFilterOption = (foCaseInsensitive, foNoPartialCompare);
  TFilterOptions = set of TFilterOption;

property FilterOptions: TFilterOptions
```

D

Description Set *FilterOptions* to specify whether or not filtering is case insensitive when filtering on string or character fields, and whether or not partial comparisons for matching filter conditions is allowed.

By default, *FilterOptions* is set to an empty set. For filters based on string fields, set *FilterOptions* to *foCaseInsensitive* to catch all variations on a string regardless of capitalization.

When a string in a filter ends with an asterisk (*), it can be used to match partial strings. To disable matching of partial strings and to treat the asterisk as a literal character in string comparisons, set *FilterOptions* to include *foNoPartialCompare*.

See also *Filter* property, *Filtered* property, *OnFilterRecord* event, *Locate* method, *Lookup* method

Found

Indicates whether or not moving to a different record is successful or not.

```
property Found: Boolean;
```

Description Check the status of *Found* to determine if a call to *FindFirst*, *FindLast*, *FindNext* or, *FindPrior* succeeds. If *Found* is *True*, success is indicated. If *False*, the move to a different record failed.

See also *Filter* property, *FindFirst* method, *FindLast* method, *FindNext* method, *FindPrior* method

Modified

Indicates whether the current record is modified.

```
property Modified: Boolean;
```

Description Check *Modified* to determine if the current record is modified. If *Modified* is *True*, the current record is modified. If *False*, the current record is not modified.

Note In general, an application need not check the status of *Modified*. Properties, events, and methods of *TDataSet* and its descendants that modify records generally check this status automatically and take appropriate actions based on its value.

See also *State* property, *Delete* method, *Edit* method, *Insert* method, *Post* method

RecNo

Implements a property that indicates the current record in the dataset.

```
property RecNo: Integer;
```

Description This function returns *-1* in all cases. Ordinarily an application will not access *RecNo* at the *TDataSet* level. Instead a redeclared and implemented *RecNo*

property in a descendent object such as *TTable* is accessed. *RecNo* provides a fallback property for derived dataset objects that do not reimplement it.

See also *RecordCount* property, *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

RecordCount

Indicates the total number of records associated with the dataset.

```
property RecordCount: Integer;
```

Description This function returns *–1* in all cases. Ordinarily an application will not access *RecordCount* at the *TDataSet* level. Instead a redeclared and implemented *RecordCount* property in a descendent object such as *TTable* is accessed. *RecordCount* provides a fallback property for derived dataset objects that do not reimplement it.

See also *RecNo* property, *RecordSize* property, *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

RecordSize

Implements a property to indicate the size of a record in the dataset.

```
property RecordSize: Word;
```

Description *RecordSize* is an abstract property that is redeclared and implemented in descendent objects derived from *TDataSet*. For *TDataSet*, *RecordSize* is meaningless.

See also *RecordCount* property, *RecNo* property, *Open* method

State

Indicates the current operating mode of the dataset.

```
type
  TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields,
    dsFilter, dsNewValue, dsOldValue, dsCurValue);

property State: TDataSetState;
```

Description Examine *State* to determine the current operating mode of the dataset. *State* determines what can be done with data in a dataset, such as editing existing records or inserting new ones. The dataset state constantly changes as an

D

application processes data. The following table lists all possible values for *State* and describes what they indicate:

Value	Meaning
dsInactive	Dataset is closed, so its data is unavailable.
dsBrowse	Data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Current record can be modified.
dsInsert	A new record can be inserted.
dsSetKey	*TTable* only. Record searching is enabled, or a *SetRange* operation is under way. A restricted set of data can be viewed, and no data can be edited or inserted.
dsCalcFields	An *OnCalcFields* event is in progress. Noncalculated fields cannot be edited, and new records cannot be inserted.
dsFilter	An *OnFilterRecord* event is in progress. A restricted set of data can be viewed. No data can edited or inserted.
dsNewValue	Temporary state used internally to indicate that the *TField.NewValue* property is being accessed.
dsOldValue	Temporary state used internally to indicate that the *TField.OldValue* property is being accessed.
dsCurValue	Temporary state used internally to indicate that the *TField.CurValue* property is being accessed.

Opening a dataset changes *State* from *dsInactive* to *dsBrowse*. An application can call *Edit* to put a dataset into *dsEdit* state, or call *Insert* to put a dataset into *dsInsert* state. If a dataset is a *TTable* object, an application can call *SetKey* or *SetRange* to put the dataset into *dsSetKey* state.

Posting or cancelling edits, insertions, or deletions, changes *State* from its current state to *dsBrowse*. Closing a dataset changes its state to *dsInactive*.

Some states, such as *dsCalcFields*, *dsFilter*, *dsNewValue*, *dsOldValue*, and *dsCurValue* cannot be seen or set directly by an application. These states are automatically set when *OnCalcField* and *OnFilterRecord* events occur, or when an application accesses the *TField.XXXValue* properties.

See also *Active* property, *CachedUpdates* property, *Filtered* property, *OnCalcFields* event, *OnFilterRecord* event, *Close* method, *Edit* method, *Insert* method, *Open* method, *TField* object, *TTable.SetKey* method, *TTable.SetRange* method

Events

The following table lists the events of *TDataSet*:

AfterCancel	public	TDataSet
AfterClose	public	TDataSet
AfterDelete	public	TDataSet
AfterEdit	public	TDataSet

AfterCancel	public	TDataSet
AfterInsert	public	TDataSet
AfterOpen	public	TDataSet
AfterPost	public	TDataSet
AfterScroll	public	TDataSet
BeforeCancel	public	TDataSet
BeforeClose	public	TDataSet
BeforeDelete	public	TDataSet
BeforeEdit	public	TDataSet
BeforeInsert	public	TDataSet
BeforeOpen	public	TDataSet
BeforePost	public	TDataSet
BeforeScroll	public	TDataSet
OnCalcFields	public	TDataSet
OnDeleteError	public	TDataSet
OnEditError	public	TDataSet
OnFilterRecord	public	TDataSet
OnNewRecord	public	TDataSet
OnPostError	public	TDataSet

AfterCancel

AfterCancel occurs after an application completes a request to cancel modifications to the current record.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterCancel: TDataSetNotifyEventl;
```

Description Write an *AfterCancel* event handler to take specific action after an application cancels changes to the current record. *AfterCancel* is called by the *Cancel* method after it updates the cursor position, releases the lock on the current record if necessary, and sets the dataset state to *dsBrowse*. If an application requires additional processing before returning control to a user after a *Cancel* event, code it in the *AfterCancel* event.

See also *State* property, *BeforeCancel* event

AfterClose

AfterClose occurs after an application closes a dataset.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterClose: TDataSetNotifyEventl;
```

D

Description Write an *AfterClose* event handler to take specific action immediately after an application closes a dataset. For example, as a security measure, an application might clear a PASSWORD entry from the *Params* property of a database component when the dataset is closed.

AfterClose is called after a dataset is closed and the dataset state is set to *dsInactive*.

See also *Active* property, *State* property, *BeforeClose* event, *Close* method, *TDatabase.Params* property

AfterDelete

AfterDelete occurs after an application deletes a record.

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterDelete: TDataSetNotifyEventl;
```

Description Write an *AfterDelete* event handler to take specific action immediately after an application deletes the current record in a dataset. *AfterDelete* is called by *Delete* after it deletes the record, sets the dataset state to *dsBrowse*, and repositions the cursor on the record prior to the one just deleted.

See also *State* property, *BeforeDelete* event, *Delete* method

AfterEdit

AfterEdit occurs after an application edits a record.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterEdit: TDataSetNotifyEventl;
```

Description Write an *AfterEdit* event handler to take specific action immediately after an application edits a record. *AfterEdit* is called by *Edit* after it enables editing of a record, recalculates calculated fields, and calls the data event handler to process a record change.

See also *BeforeEdit* event, *Edit* method

AfterInsert

AfterInsert occurs after an application inserts a new record.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterInsert: TDataSetNotifyEventl;
```

Description Write an *AfterInsert* event handler to take specific action immediately after an application inserts a record. *AfterInsert* is called by *Insert* and *Append* after inserting or appending a new record.

See also *BeforeInsert* event, *Append* method, *Insert* method

AfterOpen

AfterOpen occurs after an application completes opening a dataset and before any data access occurs.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterOpen: TDataSetNotifyEventl;
```

Description Write an *AfterOpen* event handler to take specific action immediately after an application opens the dataset. *AfterOpen* is called after the cursor for the dataset is opened and the dataset is put into *dsBrowse* state. For example, an *AfterOpen* event handler might check the system registry to determine the last record touched in the dataset the previous time the application ran, and position the cursor at that record.

See also *Active* property, *State* property, *BeforeOpen* event, *Open* method

AfterPost

AfterPost occurs after an application writes the current record to the database or cache and before the dataset is returned to browse state.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterPost: TDataSetNotifyEventl;
```

Description Write an *AfterPost* event handler to take specific action immediately after an application posts a change to the current record. *AfterPost* is called after a modification, deletion, or insertion is made to a record.

See also *BeforePost* event, *Post* method

AfterScroll

AfterScroll occurs after an application scrolls from one record to another.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property AfterScroll: TDataSetNotifyEventl;
```

Description Write an *AfterScroll* event handler to take specific action immediately after an application scrolls to another record as a result of a call to the *First*, *Last*, *MoveBy*, *Next*, *Prior*, *FindKey*, *FindFirst*, *FindNext*, *FindLast*, *FindPrior*, and *Locate* methods. *AfterScroll* is called after all other events triggered by these methods and any other methods that switch from record to record in the dataset.

See also *BeforeScroll* event, *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

BeforeCancel

BeforeCancel occurs before an application executes a request to cancel changes to the current record.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforeCancel: TDataSetNotifyEventl;
```

Description Write a *BeforeCancel* event to take specific action before an application carries out a request to cancel changes. *BeforeCancel* is called by the *Cancel* method before it cancels a dataset operation such as *Edit*, *Insert*, or *Delete*.

An application might use the *BeforeCancel* event to record a user's changes in an undo buffer.

See also *AfterCancel* event, *Delete* method, *Edit* method, *Insert* method

BeforeClose

BeforeClose occurs before an application executes a request to close the dataset.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforeClose: TDataSetNotifyEventl;
```

Description Write a *BeforeClose* event to take specific action before an application closes a dataset. Calling *Close* or setting the *Active* property to *False* results in a call to the *BeforeClose* event handler.

See also *Active* property, *AfterClose* event, *Close* method

BeforeDelete

BeforeDelete occurs before an application attempts to delete the current record.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforeDelete: TDataSetNotifyEventl;
```

Description Write a *BeforeDelete* event handler to take specific action before an application deletes the current record. *BeforeDelete* is called by *Delete* before it actually deletes a record.

Making use of this event an application might, for example, display a dialog box asking for confirmation before deleting the record. On denial of

confirmation, the application could abort the deletion by calling the *Abort* procedure.

See also *AfterDelete* event, *Delete* method, *Abort* procedure

BeforeEdit

BeforeEdit occurs before an application enters edit mode for the current record.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforeEdit: TDataSetNotifyEventl;
```

Description Write a *BeforeEdit* event handler to take specific action before an application enables editing of the current record. For example, an application might keep a log of database edits, and therefore might record the edit request, time, and user in a *BeforeEdit* event before entering edit state.

See also *State* property, *AfterEdit* event, *Edit* method

BeforeInsert

BeforeInsert occurs before an application enters insert mode.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforeInsert: TDataSetNotifyEventl;
```

Description Write a *BeforeInsert* event handler to take specific action before an application inserts or appends a new record. *BeforeInsert* is called by *Insert* or *Append* before they set the dataset into dsInsert state.

See also *State* property, *AfterInsert* event, *Append* method, *Insert* method

BeforeOpen

BeforeOpen occurs before an application executes a request to open a dataset.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforeOpen: TDataSetNotifyEventl;
```

Description Write a *BeforeOpen* event handler to take specific action before an application opens a dataset for viewing or editing. *BeforeOpen* is triggered when an application sets the *Active* property to *True* for a dataset or an application calls *Open*.

See also *Active* property, *AfterOpen* event, *Open* method

BeforePost

BeforePost occurs before an application posts changes for the current record to the database or cache.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforePost: TDataSetNotifyEventl;
```

Description Write a *BeforePost* event handler to take specific action before an application posts dataset changes to the database. *BeforePost* is triggered when an application calls the *Post* method. *Post* checks to make sure all required fields are present, then calls *BeforePost* before posting the record.

An application might use *BeforePost* to perform validity checks on data changes before posting them to the database. If it encountered a validity problem, it could call *Abort* to cancel the *Post* operation.

See also *AfterPost* event, *Post* method, *Abort* procedure

BeforeScroll

BeforeScroll occurs before an application scrolls from one record to another.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property BeforeScroll: TDataSetNotifyEventl;
```

Description Write a *BeforeScroll* event handler to take specific action immediately after an application scrolls to another record as a result of a call to the *First*, *Last*, *MoveBy*, *Next*, *Prior*, *FindKey*, *FindFirst*, *FindNext*, *FindLast*, *FindPrior*, and *Locate* methods. *BeforeScroll* is called before all other events triggered by these methods and any other methods that switch from record to record in the dataset.

See also *BeforeScroll* event, *First* method, *Last* method, *MoveBy* method, *Next* method, *Prior* method

OnCalcFields

OnCalcFields occurs when an application recalculates calculated fields.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property OnCalcFields: TDataSetNotifyEventl;
```

Description Write an *OnCalcFields* event handler to take specific action when an application recalculates calculated fields. A calculated field is one that derives its value from the values in one or more fields in the dataset, sometimes with additional processing.

OnCalcFields is called frequently, so the code it contains should be brief.

When the *AutoCalcFields* property is *True*, *OnCalcFields* is triggered when:

- A dataset is opened.
- A dataset is put into *dsEdit* state.
- Focus moves from one visual control to another, or from one column to another is a data-aware grid control and modifications were made to the record.
- A record is retrieved from a database.

Note When the *AutoCalcFields* property is *True*, *OnCalcFields* should not modify the dataset (or a linked dataset if it is part of a master-detail relationship), because such modifications retrigger the *OnCalcField* event, leading to recursion.

To reduce the frequency with which *OnCalcFields* is called, set *AutoCalcFields* to *False*. When *AutoCalcFields* is *False*, *OnCalcFields* is not called when changes are made to individual fields within a record. If an application permits users to change data, *OnCalcFields* is frequently triggered. In these cases an application may set *AutoCalcFields* to *False* to reduce the frequency with which *AutoCalcFields* is called.

See also *Active* property, *AutoCalcFields* property, *Open* method, *Post* method

OnDeleteError

OnDeleteError occurs when an application attempts to delete a record and an exception is raised.

```
type
  TDataAction = (daFail, daAbort, daRetry);
  TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
    var Action: TDataAction) of object;

property OnDeleteError: TDataSetErrorEven;
```

Description Write an *OnDeleteError* event handler to handle exceptions that occur when an attempt to delete a record fails. *E* is a pointer to the database error object that contains the exception error message so that an application can display an error message. *Action* indicates how the handler should respond to the error. The following table summarizes the settings for *Action* and what they mean:

Value	Meaning
daFail	Aborts the delete operation and displays an error message.
daAbort	Aborts the delete operation without displaying a message.
daRetry	Repeats the delete operation. Correct the error condition inside the error handler before setting Action to daRetry.

When *OnDeleteError* is first invoked, *Action* is always set to *daFail*. If the error handler can correct the error condition that caused the handler to be invoked, set *Action* to *daRetry* before exiting the handler. When *Action* is *daRetry*, the delete operation is tried again.

If an error condition cannot be corrected, the display of the error message can be suppressed, if desired, by setting *Action* to *daAbort* instead of *daFail*.

See also *OnEditError* event, *OnPostError* event, *Delete* method

OnEditError

OnEditError occurs when an application attempts to modify or insert a record and an exception is raised.

```
type
  TDataAction = (daFail, daAbort, daRetry);
  TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
    var Action: TDataAction) of object;

property OnEditError: TDataSetErrorEven;
```

Description Write an *OnEditError* event handler to handle exceptions that occur when an attempt to edit a record fails. *E* is a pointer to the database error object that contains the exception error message so that an application can display an error message. *Action* indicates how the handler should respond to the error. The following table summarizes the settings for *Action* and what they mean:

Value	Meaning
daFail	Aborts the edit operation and displays an error message.
daAbort	Aborts the edit operation without displaying a message.
daRetry	Repeats the edit operation. Correct the error condition inside the error handler before setting Action to daRetry.

When *OnEditError* is first invoked, *Action* is always set to *daFail*. If the error handler can correct the error condition that caused the handler to be invoked, set *Action* to *daRetry* before exiting the handler. When *Action* is *daRetry*, the edit operation is tried again.

If an error condition cannot be corrected, the display of the error message can be suppressed, if desired, by setting *Action* to *daAbort* instead of *daFail*.

See also *OnDeleteError* event, *OnPostError* event, *Edit* method, *Insert* method

OnFilterRecord

OnFilterRecord occurs each time a different record in the dataset becomes the current record and filtering is enabled.

```
type
  TFilterRecordEvent = procedure(DataSet: TDataSet; var Accept: Boolean) of object;
```

```
property OnFilterRecord: TFilterRecordEvent;
```

Description Write an *OnFilterRecord* event handler to test each record in a dataset for against a test condition that determines whether or not the record is visible to the application.

To indicate whether or not a record passes the filter condition, a filter handler must set the *Accept* parameter to *True* to include a record, or *False* to exclude it. For example, the following filter displays only those records with a State field set to "CA":

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'] = 'CA';
end;
```

Filtering is enabled if the *Filtered* property is *True*. When an application is processing a filter, the *State* property for the dataset is *dsFilter*. The *FilterOptions* property controls case sensitivity and filtering on partial comparisons.

Note Applications can also specify a filter using the *Filter* property. The *Filter* property supplements the *OnFilterRecord* event handler. Be sure that the interactions between the *Filter* property and the *OnFilterRecord* event handler do not result in an empty filter set when they are used simultaneously in an application.

See also *Filter* property, *Filtered* property, *FilterOptions* property, *State* property

OnNewRecord

OnNewRecord occurs when an application inserts or appends a new dataset record.

```
type
  TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;

property OnNewRecord: TDataSetNotifyEventl;
```

Description Write an *OnNewRecord* event handler to take specific actions as an application inserts or appends a new record. *OnNewRecord* is called as part of the actual insert or append process. An application might use the *OnNewRecord* event to set initial values for a record or as a way of implementing cascading insertions in related datasets.

See also *AfterInsert* event, *BeforeInsert* event

OnPostError

OnPostError occurs when an application attempts to modify or insert a record and an exception is raised.

D

```
type
  TDataAction = (daFail, daAbort, daRetry);
  TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
    var Action: TDataAction) of object;

property OnPostError: TDataSetErrorEven;
```

Description Write an *OnPostError* event handler to handle exceptions that occur when an attempt to edit a record fails. *E* is a pointer to the database error object that contains the exception error message so that an application can display an error message. *Action* indicates how the handler should respond to the error. The following table summarizes the settings for *Action* and what they mean:

Value	Meaning
daFail	Aborts the post operation and displays an error message.
daAbort	Aborts the post operation without displaying a message.
daRetry	Repeats the post operation. Correct the error condition inside the error handler before setting Action to daRetry.

When *OnPostError* is first invoked, *Action* is always set to *daFail*. If the error handler can correct the error condition that caused the handler to be invoked, set *Action* to *daRetry* before exiting the handler. When *Action* is *daRetry*, the edit operation is tried again.

If an error condition cannot be corrected, the display of the error message can be suppressed, if desired, by setting *Action* to *daAbort* instead of *daFail*.

See also *OnDeleteError* event, *OnEditError* event, *Post* method

Methods

The following table lists the methods of *TDataSet*:

Method	Visibility	Documented in
ActiveBuffer	public	TDataSet
Append	public	TDataSet
AppendRecord	public	TDataSet
Assign	public	TPersistent
BookmarkValid	public	TDataSet
Cancel	public	TDataSet
CheckBrowseMode	public	TDataSet
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject

Method	Visibility	Documented in
ClearFields	public	TDataSet
Close	public	TDataSet
CompareBookmarks	public	TDataSet
ControlsDisabled	public	TDataSet
Create	public	TDataSet
CursorPosChanged	public	TDataSet
DefaultHandler	public	TObject
Delete	public	TDataSet
Destroy	public	TDataSet
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableControls	public	TDataSet
Dispatch	public	TObject
Edit	public	TDataSet
EnableControls	public	TDataSet
FieldAddress	public	TObject
FieldByName	public	TDataSet
FindComponent	public	TComponent
FindField	public	TDataSet
FindFirst	public	TDataSet
FindLast	public	TDataSet
FindNext	public	TDataSet
FindPrior	public	TDataSet
First	public	TDataSet
Free	public	TObject
FreeBookmark	public	TDataSet
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetBookmark	public	TDataSet
GetCurrentRecord	public	TDataSet
GetFieldList	public	TDataSet
GetFieldNames	public	TDataSet
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GotoBookmark	public	TDataSet
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject

Method	Visibility	Documented in
Insert	public	TDataSet
InsertComponent	public	TComponent
InsertRecord	public	TDataSet
InstanceSize	public	TObject
IsEmpty	public	TDataSet
IsLinkedTo	public	TDataSet
Last	public	TDataSet
Locate	public	TDataSet
Lookup	public	TDataSet
MethodAddress	public	TObject
MethodName	public	TObject
MoveBy	public	TDataSet
NewInstance	public	TObject
Next	public	TDataSet
Open	public	TDataSet
Post	public	TDataSet
Prior	public	TDataSet
Refresh	public	TDataSet
RemoveComponent	public	TComponent
Resync	public	TDataSet
SafeCallException	public	TComponent
SetFields	public	TDataSet
Translate	public	TDataSet
UpdateCursorPos	public	TDataSet
UpdateRecord	public	TDataSet

ActiveBuffer

Returns a pointer to the buffer for the current record.

```
function ActiveBuffer: PChar;
```

Description

ActiveBuffer is used internally by many dataset methods to ensure that the active buffer points to the buffer for the current record. If an application uses existing dataset methods, the active buffer is always set correctly, so there is usually no need to call *ActiveBuffer* directly.

ActiveBuffer is also used by bookmarking methods to index into the current record buffer to retrieve bookmark information.

Applications that provide custom dataset routines may need to call *ActiveBuffer* to access the buffer data, especially when iterating through a set of records in the dataset. Even in these cases, if custom routines call methods such as *Next* or *Prior*, the active buffer is automatically set.

See also

RecordSize property, *Next* method, *Prior* method

Append

Adds a new, empty record to the end of the dataset.

```
procedure Append;
```

Description Call *Append* to:

1. Open a new, empty record at the end of the dataset.
2. Set the current record to the new record.

After a call to *Append*, an application can enable users to enter data in the fields of the record, and can then post those changes to the database using *Post* (or *ApplyUpdates* if cached updating is enabled). A newly appended record is posted to the database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. For indexed tables, the index is updated with the new record information.

See also *AppendRecord* method, *ApplyUpdates* method, *Insert* method, *Post* method

AppendRecord

Adds a new, populated record to the end of the dataset and posts it to the database.

```
procedure AppendRecord(const Values: array of const);
```

Description Call *AppendRecord* to create a new, empty record at the end of the dataset, populate it with the field values in *Values*, and post the values to the database. A newly appended record is posted to the database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. For indexed tables, the index is updated with the new record information.

The newly appended record becomes the current record.

See also *Append* method, *InsertRecord* method, *Post* method, *SetFields* method

BookmarkValid

Implements a virtual function that tests the validity of a specified bookmark.

```
type
  TBookmark = Pointer;

function BookmarkValid(Bookmark: TBookmark): Boolean; virtual;
```

Description This function returns *False* in all cases. Ordinarily an application should not call *BookmarkValid* at the *TDataSet* level, but instead call a redeclared and implemented *BookmarkValid* in a descendent object such as *TTable*. *BookmarkValid* provides a fallback method for derived dataset objects that do not reimplement *BookmarkValid*.

See also *CompareBookmarks* method, *TTable* object

Cancel

Cancels modifications to the current record if those changes are not yet posted.

```
procedure Cancel; virtual;
```

Description Call *Cancel* to undo modifications made to one or more fields belonging to the current record. As long as those changes are not already posted to the database, *Cancel* returns the record to its previous state, and sets the dataset state to *dsBrowse*.

Typically *Cancel* is used to back out of changes in response to user request, or in field validation routines that back out illegal field values. The *TDBNavigator* object contains a Cancel button that triggers a call to *Cancel*.

See also *State* property, *Post* method, *TDBNavigator* object

CheckBrowseMode

Automatically posts or cancels data changes when an application calls a method that changes which record in a dataset is the current record.

```
procedure CheckBrowseMode;
```

Description *CheckBrowseMode* is used internally by many dataset methods to ensure that modifications to the current record are posted to the database when a dataset's state is *dsEdit*, *dsInsert*, or *dsSetKey* state and a method is called that switches to a different record.

If *State* is *dsEdit* or *dsInsert*, *CheckBrowseMode* calls *UpdateRecord*, and then, if the *Modified* property for the dataset is *True*, *CheckBrowseMode* calls *Post*. If *Modified* is *False*, *CheckBrowseMode* calls *Cancel*.

If *State* is *dsSetKey*, *CheckBrowseMode* calls *Post*.

If *State* is *dsInactive*, *CheckBrowseMode* raises an exception.

If an application uses existing dataset methods, *CheckBrowseMode* is always called when necessary, so there is usually no need to call *CheckBrowseMode* directly.

Applications that provide custom dataset routines may need to call *CheckBrowseMode* inside those routines to guarantee that changes are posted to the database when switching to a different record.

See also *State* property, *Cancel* method, *Edit* method, *First* method, *Last* method, *MoveBy* method, *Next* method, *Post* method, *Prior* method, *TTable.SetKey* method

ClearFields

Clears the contents of all fields for the current record.

```
procedure ClearFields;
```

Description Call *ClearFields* to erase the current contents of all fields for the current record. If the dataset is not in either *dsInsert* or *dsEdit* state, *ClearFields* raises an exception. Otherwise it calls the *DataEvent* event handler to invoke *CheckBrowseMode*, and then initialize all fields for the record, effectively clearing them. If a *SetKey* operation is not under way, *ClearFields* recalculates all calculated fields, and calls the *DataEvent* handler with a parameter indicating that the record has changed. The *DataEvent* handler, in turn, calls the *OnDataChange* event handler for the data source component associated with the dataset.

See also *CheckBrowseMode* method, *DataEvent* event, *TTable.SetKey* method, *TDataSource.OnDataChange* event

Close

Closes a dataset.

```
procedure Close;
```

Description Call *Close* to set the *Active* property of a dataset to *False*. When *Active* is *False*, the dataset is closed; the dataset cannot read data from or write data to the database.

An application must set *Active* to *False* before changing other properties that affect the status of the database or the controls that display data in an application. For example, to change the *DataSource* property for a dataset, *Active* must first be set to *False*, ensuring that the dataset is closed. Setting *Active* to *False* puts the dataset into dsInactive state and closes the BDE cursor.

See also *Active* property, *DataSource* property, *State* property, *AfterClose* event, *BeforeClose* event, *Open* method

CompareBookmarks

Implements a virtual function to compare two bookmarks.

```
type
  TBookmark = Pointer;

function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer; virtual;
```

Description This function returns *0* in all cases. Ordinarily an application should not call *CompareBookmarks* at the *TDataSet* level, but instead call a redeclared and implemented *CompareBookmarks* in a descendent object such as *TTable*. *CompareBookmarks* provides a fallback method for derived dataset objects that do not reimplement *CompareBookmarks*.

See also *BookmarkValid* method, *TTable* object

ControlsDisabled

Determines if the updating of data display in data-aware controls is currently disabled for the dataset.

```
function ControlsDisabled: Boolean;
```

Description Call *ControlsDisabled* to determine if the updating of data display in data-aware controls is currently disabled. If *ControlsDisabled* is *True*, controls are currently disabled. *ControlsDisabled* is *True* as long as the reference count that keeps track of disabling for the dataset is greater than zero. This count is incremented every time the *DisableControls* procedure is called. Applications should call *DisableControls* to improve performance and prevent constant display updating during an automated iteration through a large number of records in the dataset.

In complex applications, when controls may be disabled multiple times by different processes, you can use *ControlsDisabled* as a check in a procedure to re-enable controls should each call to *DisableControls* not be paired with a subsequent call to *EnableControls*. For example,

```
procedure ReEnableControls (DataSet: TDataSet);
begin
  while DataSet.ControlsDisabled do
    DataSet.EnableControls;
end;
```

See also *DisableControls* method, *EnableControls* method

Create

Creates an instance of a *TDataSet* component.

```
constructor Create(AOwner: TComponent);
```

Description Call *Create* to instantiate a dataset component at runtime. Ordinarily applications instantiate dataset descendants, such as *TTable*, *TQuery*, or

TStoredProc, rather than *TDataSet*, and these instantiated objects are automatically handled by Delphi.

On the other hand, applications that create specialized dataset components, such as custom components that bypass the Borland Database Engine (BDE) may need to instantiate a *TDataSet* component by calling *Create*. *Create*:

- Creates empty field definitions for the dataset.
- Creates an empty list of fields for the dataset.
- Creates an empty list of data sources for the dataset.
- Set the *AutoCalcFields* property to *True*.
- Clears the field buffers for the dataset.

See also *AutoCalcFields* property, *Locale* property, *Destroy* method, *TQuery* object, *TStoredProc* object, *TTable* object

CursorPosChanged

Sets the cursor position in the dataset to –1.

```
procedure CursorPosChanged;
```

Description *CurPosChanged* is an internal method that invalidates the variable that tracks the physical cursor position relative to the logical cursor position. *CurPosChanged* is called by the *Locate* and *Lookup* methods prior to searching for a requested record. These methods, if successful, reposition the cursor at the first matching record found.

See also *Locate* method, *Lookup* method

Delete

Deletes the current record and positions the cursor on the next record.

```
procedure Delete;
```

Description Call *Delete* to remove the current record from the database. If the dataset is inactive, *Delete* raises an exception. Otherwise *Delete*:

- Verifies that the dataset is not empty (and raises an exception if it is).
- Calls *CheckBrowseMode* to post any pending changes to a prior record if necessary.
- Triggers the *BeforeDelete* event handler.
- Deletes the record.
- Frees the buffers allocated for the record.
- Puts the dataset into *dsBrowse* mode.
- Resynchronizes the dataset to position the cursor on the next undeleted record.
- Calls the *AfterDelete* event handler.

See also *State* property, *AfterDelete* event, *BeforeDelete* event, *CheckBrowseMode* method, *Edit* method, *Insert* method, *Post* method

Destroy

Destroys the instance of a dataset component.

```
destructor Destroy;
```

Description Do not call *Destroy* directly in an application. Usually destruction of datasets is handled automatically by Delphi. If an application creates its own instances of a dataset, however, the application should call *Free*, which verifies that the dataset component is not already freed before calling *Destroy*.

Destroy performs the following tasks:

- Closes the dataset.
- Frees the Dataset Designer for the dataset, if one exists.
- Disassociated the dataset from its data source components and frees the memory allocated for them.
- Destroys the fields associated with the dataset and frees the memory allocated to them.
- Frees the memory allocated for field definitions.
- Calls the inherited *Destroy* destructor.

See also *Designer* property, *FieldDefs* property, *UpdateObject* property, *Create* method

DisableControls

Disables data display in data-aware controls associated with the dataset through a data source component.

```
procedure DisableControls;
```

Description Call *DisableControls* to prevent data display in data-aware controls prior to iterating through a large number of records in the dataset. Disabling controls prevents flicker in controls during rapid iteration through the dataset and its speeds performance because data does not need to be written to the display.

If controls are not already disabled, *DisableControls* records the current state of the dataset, sets the enabling event for the dataset to *deDataSetChange*, and increments the dataset's disabled count variable. Otherwise, *DisableControls* just increments the disabled count variable.

The enabling event and disabled count variable are used internally by other methods and objects to determine whether or not to display data in data-aware controls. When the disable count variable is greater than zero, data is not displayed.

Usually *DisableControls* is called within the context of a **try...finally** block that re-enables the controls even if an exception occurs. For example,

```
⋮
with CustTable do
begin
  DisableControls;
  try
    First;
    while not EOF do
    begin
      { Process each record here }
      ⋮
      Next;
    end;
  finally
    EnableControls;
  end;
end;
```

Note *DisableControls* can safely be called when controls are already disabled. In complex applications there may be separate operations that are sometimes nested, both of which need to disable controls.

See also *ControlsDisabled* method, *EnableControls* method

Edit

Enables editing of data in the dataset.

procedure Edit;

Description Call *Edit* to permit editing of the current record in a dataset. *Edit* determines the current state of the dataset. If the dataset is empty, *Edit* calls *Insert*. Otherwise *Edit*:

- Calls *CheckBrowseMode* to post any pending changes to a prior record if necessary.
- Triggers the *BeforeEdit* event handler.
- Retrieves the record.
- Puts the dataset into *dsEdit* state, enabling the application or user to modify fields in the record.
- Triggers the record change event.
- Triggers the *AfterEdit* event handler.

See also *State* property, *AfterEdit* event, *BeforeEdit* event, *CheckBrowseMode* method, *Delete* method, *Insert* method

D

EnableControls

Enables data display in data-aware controls associated with the dataset through a data source component.

```
procedure EnableControls;
```

Description Call *EnableControls* to permit data display in data-aware controls after a prior call to *DisableControls*. *EnableControls* decrements the disabled count variable for the dataset if it is not already zero. If the disable count variable is zero, *EnableControls* updates the current state of the dataset, if necessary, and then call an event handler for re-enabling data display.

Usually *EnableControls* is called within the context of a **try...finally** block to re-enables controls even if an exception occurs while controls are disabled. For example,

```
  ⋮
  with CustTable do
  begin
    DisableControls;
    try
      First;
      while not EOF do
      begin
        { Process each record here }
        ⋮
        Next;
      end;
    finally
      EnableControls;
    end;
  end;
```

See also *ControlsDisabled* method, *DisableControls* method

FieldByName

Finds a field based on a specified field name.

```
function FieldByName(const FieldName: string): TField;
```

Description Call *FieldByName* to retrieve field information for a field when only the field's name is known. *FieldName* is the name of an existing field. *FieldByName* returns the *TField* component for the specified field.

An application can access specific properties and methods of the field directly. For example, the following statement determines if a specified field is a calculated field or not:

```
if Customers.FieldByName('FullName').Calculated then
  MessageDlg(Format('%s is a calculated field. ', ['FullName']), mtInformation,
[mbOK], 0);
```

FieldByName is especially useful at design time for developers who are creating database applications, but who do not have access to the underlying table and therefore cannot use persistent field components.

Note To retrieve or set the value for a specific field, call the default dataset method *FieldValues* instead of *FieldByName*.

See also *Fields* property, *FindField* method, *FieldValues* method, *TField* object

FindField

Searches for a specified field object in the dataset.

```
function FindField(const FieldName: string): TField;
```

Description Call *FindField* to determine if a specified field component exists in a dataset. *FieldName* is the name of the field for which to search. If *FindField* finds a field with a matching name, it returns the *TField* component for the specified field. Otherwise it returns **nil**.

FindField is a useful function to call prior to calling other dataset methods, such as *FieldByName*, that require a valid field name as a parameter.

See also *Fields* property, *FieldByName* method

FindFirst

Implements a virtual method for positioning the cursor on the first record in a filtered dataset.

```
function FindFirst: Boolean;
```

Description This function returns *False*, indicating that the cursor was not successfully repositioned. Ordinarily an application should not call *FindFirst* at the *TDataSet* level, but instead call a redeclared and implemented *FindFirst* in a descendent object such as *TTable*. *FindFirst* provides a fallback method for derived dataset objects that do not reimplement *FindFirst*.

See also *Filter* property, *Filtered* property, *OnFilterRecord* event, *FindLast* method, *FindNext* method, *FindPrior* method

FindLast

Implements a virtual method for positioning the cursor on the last record in a filtered dataset.

```
function FindLast: Boolean;
```

Description This function returns *False*, indicating that the cursor was not successfully repositioned. Ordinarily an application should not call *FindLast* at the *TDataSet* level, but instead call a redeclared and implemented *FindLast* in a descendent object such as *TTable*. *FindLast* provides a fallback method for derived dataset objects that do not reimplement *FindLast*.

See also *Filter* property, *Filtered* property, *OnFilterRecord* event, *FindFirst* method, *FindNext* method, *FindPrior* method

FindNext

Implements a virtual method for positioning the cursor on the next record in a filtered dataset.

```
function FindNext: Boolean;
```

Description This function returns *False*, indicating that the cursor was not successfully repositioned. Ordinarily an application should not call *FindNext* at the *TDataSet* level, but instead call a redeclared and implemented *FindNext* in a descendent object such as *TTable*. *FindNext* provides a fallback method for derived dataset objects that do not reimplement *FindNext*.

See also *Filter* property, *Filtered* property, *OnFilterRecord* event, *FindFirst* method, *FindLast* method, *FindPrior* method

FindPrior

Implements a virtual method for positioning the cursor on the previous record in a filtered dataset.

```
function FindPrior: Boolean;
```

Description This function returns *False*, indicating that the cursor was not successfully repositioned. Ordinarily an application should not call *FindPrior* at the *TDataSet* level, but instead call a redeclared and implemented *FindPrior* in a descendent object such as *TTable*. *FindPrior* provides a fallback method for derived dataset objects that do not reimplement *FindPrior*.

See also *Filter* property, *Filtered* property, *OnFilterRecord* event, *FindFirst* method, *FindLast* method, *FindNext* method

First

Implements a generic method to position the cursor on the first record in the dataset.

```
procedure First;
```

Description Do not call *First* at the *TDataSet* level. In *TDataSet* descendents, call *First* to position the cursor on the first record in the dataset and make it the current record. *First* posts any changes to the current record and:

- Clears the record buffers.
- Sets the cursor to the beginning of the dataset file.
- Calls a **virtual, abstract** method to fetch the first record, position the cursor on it, and make it the current record. For *TDataSet* this method is only declared, not implemented.

- Fetches any additional records required for display, such as those needed to fill out a grid control. In *TDataSet* this method calls another **virtual**, **abstract** method that is only declared, not implemented.
- Sets the *BOF* property to *True*.
- Triggers an event handler for dataset change events.

See also *Last* method, *MoveBy* method, *Next* method, *Prior* method

FreeBookmark

Frees the resources allocated for a specified bookmark.

```
type
  TBookmark = Pointer;

procedure FreeBookmark(Bookmark: TBookmark); virtual;
```

Description Call *FreeBookmark* to:

- Free an existing bookmark before reassigning it.
- Release the memory allocated for a specified bookmark when the bookmark is no longer needed.

See also *Bookmark* property, *GetBookmark* method, *GotoBookmark* method

GetBookmark

Allocates a bookmark for the current cursor position in the dataset.

```
type
  TBookmark = Pointer;

function GetBookmark: TBookmark; virtual;
```

Description Call *GetBookmark* to establish a bookmark for the current record in the dataset. Establishing a bookmark for a record enables an application to return to that record in the dataset at any time while the bookmark exists.

GetBookmark requires that a variable of type *TBookmark* already be declared in an application. Use *GetBookmark* to assign the variable a value that can be referenced by subsequent calls to *GotoBookmark* and *FreeBookmark*.

Note Applications that create bookmarks with *GetBookmark* should subsequently release the system resource allocated to them by calling *FreeBookmark* when the bookmarks are no longer needed.

See also *FreeBookmark* method, *GotoBookmark* method

GetCurrentRecord

Implements a virtual function to retrieve the current record in a buffer.

```
function GetCurrentRecord(Buffer: PChar): Boolean; virtual;
```

Description This function always returns *False*, indicating failure. Most applications should not call *GetCurrentRecord* at the *TDataSet* level. Instead they should call a redeclared and reimplemented *GetCurrentRecord* method in a descendent object, such as *TTable*. *GetCurrentRecord* provides a fallback function for derived objects that do not redeclare and implement this function.

See also *ActiveBuffer* method

GetFieldList

Retrieves a specified set of field objects into a list.

```
procedure GetFieldList(List: TList; const FieldNames: string);
```

Description Call *GetFieldList* to copy a specified set of field objects into a list object created and maintained by the application.

List is the *TList* object into which to copy the field objects. *FieldNames* is a string containing the name of the fields to copy. Each field name in the string must be separated from other field names with a semicolon. *GetFieldList* builds a list that contains only the field objects for which it finds a matching name in the dataset's list of field objects.

Note Applications do not normally have to call *GetFieldList* to copy field objects. Field objects are directly accessible through the dataset itself. In some cases, however, it can be useful to work with a copy of a field object or its data instead of working on the actual object in the dataset. In these cases, *GetFieldList* is useful.

See also *FieldDefs* property, *Fields* property, *FieldByName* method, *FindField* method, *GetFieldNames* method

GetFieldNames

Retrieves a list of names for all fields in a dataset.

```
procedure GetFieldNames(List: TStrings);
```

Description Call *GetFieldNames* to get a list of names for all fields in a dataset. *List* is a *TStrings* object created and maintained by the application.

Retrieving a list of field names is especially useful for applications that work with datasets whose field objects are created dynamically at runtime. By retrieving a list of field names, the application can be restricted to working only with fields that exist at runtime.

See also *FieldDefs* property, *Fields* property, *GetFieldList* method

GotoBookmark

Implements a virtual method to position the cursor on the record pointed to by a specified bookmark.

```
type
  TBookmark = Pointer;

procedure GotoBookmark(Bookmark: TBookmark);
```

Description Do not call *GotoBookmark* at the *TDataSet* level. *GotoBookmark* calls a virtual, abstract internal method that is not implemented by *TDataSet*. Instead, descendents of *TDataSet* redeclare and implement the *GotoBookmark* method, and applications should call the descendents' methods instead.

See also *Bookmark* property, *FreeBookmark* method, *GetBookmark* method

Insert

Inserts a new, empty record in the dataset.

```
procedure Insert;
```

Description Call *Insert* to:

1. Open a new, empty record in the dataset.
2. Set the current record to the new record.

After a call to *Insert*, an application can enable users to enter data in the fields of the record, and can then post those changes to the database using *Post* (or *ApplyUpdates* if cached updating is enabled). A newly inserted record is posted to the database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is inserted into the dataset at the current cursor position.
- For SQL databases, the physical location of the append is implementation-specific. For indexed tables, the index is updated with the new record information.

See also *Append* method, *ApplyUpdates* method, *InsertRecord* method, *Post* method

InsertRecord

Inserts a new, populated record to the dataset and posts it to the database.

```
procedure InsertRecord(const Values: array of const);
```

Description Call *InsertRecord* to create a new, empty record at in the dataset, populate it with the field values in *Values*, and post the values to the database. A newly inserted record is posted to the database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is inserted into the dataset at the current cursor position.

D

- For SQL databases, the physical location of the insert is implementation-specific. For indexed tables, the index is updated with the new record information.

The newly inserted record becomes the current record.

See also *AppendRecord* method, *Insert* method, *Post* method, *SetFields* method

IsEmpty

Determines if a dataset contains no records.

```
function IsEmpty: Boolean;
```

Description Call *IsEmpty* to determine if a dataset has records. *IsEmpty* returns *True* if the dataset does not contain any records. Otherwise it returns *False*.

See also *BOF* property, *EOF* property

IsLinkedTo

Checks that a dataset is linked to a specified data source.

```
function IsLinkedTo(DataSource: TDataSource): Boolean;
```

Description Call *IsLinkedTo* to verify that a dataset is linked to a specific data source. *DataSource* is the name of the data source against which to test.

Note *IsLinkedTo* is mainly provided for developers deriving custom components based on *TDataSet*. It is not intended or needed for general data access.

If the dataset already uses the specified data source, *IsLinkedTo* returns *True*. If the dataset does not use the specified data source, and the data source does not already have a dataset of its own, *IsLinkedTo* returns *False*.

See also *DataSource* property

Last

Implements a generic method to position the cursor on the last record in the dataset.

```
procedure Last;
```

Description Do not call *Last* at the *TDataSet* level. In *TDataSet* descendents, call *Last* to position the cursor on the last record in the dataset and make it the current record. *Last* posts any changes to the current record and:

- Clears the record buffers.
- Sets the cursor to the end of the dataset file.
- Calls a **virtual, abstract** method to fetch the last record, position the cursor on it, and make it the current record. For *TDataSet* this method is only declared, not implemented.

- Fetches any additional records required for display, such as those needed to fill out a grid control. In *TDataSet* this method calls another **virtual, abstract** method that is only declared, not implemented.
- Sets the *EOF* property to *True*.
- Triggers an event handler for dataset change events.

See also *First* method, *MoveBy* method, *Next* method, *Prior* method

Locate

Implements a virtual method for searching a dataset for a specified record and makes that record the current record.

```
type
  TLocateOption = (loCaseInsensitive, loPartialKey);
  TLocateOptions = set of TLocateOption;
function Locate(const KeyFields: string; const KeyValues: Variant;
    Options: TLocateOptions): Boolean; virtual;
```

Description This function returns *False*, indicating that a matching record was not found and the cursor is not repositioned. Ordinarily an application does not call *Locate* at the *TDataSet* level, but instead calls a redeclared and implemented *FindLast* in a descendent object such as *TTable*. *Locate* provides a fallback method for derived dataset objects that do not reimplement *Locate*.

See also *FindField* method, *Lookup* method

Lookup

Implements a virtual method to retrieve field values from a record that matches specified search values.

```
function Lookup(const KeyFields: string; const KeyValues: Variant;
    const ResultFields: string): Variant; virtual;
```

Description This function returns *False*, indicating that a matching field was not found. Ordinarily an application does not call *Lookup* at the *TDataSet* level, but instead call a redeclared and implemented *Lookup* in a descendent object such as *TTable*. *Lookup* provides a fallback method for derived dataset objects that do not reimplement *Lookup*.

See also *Locate* method

MoveBy

Positions the cursor on a record relative to the current record in the dataset.

```
function MoveBy(Distance: Integer): Integer;
```

Description Call *MoveBy* to position the cursor on a record relative to the current record in the dataset. *Distance* indicates the number of records to move. A positive

value for *Distance* indicates forward progress through the dataset, while a negative value indicates backward progress. For example, the following statement moves backward through the dataset by 10 records:

```
MoveBy(-10);
```

MoveBy posts any changes to the current record and:

- Sets the *BOF* and *EOF* properties to *False*.
- If *Distance* is positive, repeatedly fetches subsequent records, if possible, decrementing *Distance* until it is zero, positions the cursor on the last record fetched, and makes it the current record. If an attempt is made to move past the end of the file, *MoveBy* sets *EOF* to *True*.
- If *Distance* is negative, repeatedly fetches previous records, if possible, incrementing *Distance* until it is zero, positions the cursor on the last record fetched, and makes it the current record. If an attempt is made to move past the start of the file, *MoveBy* sets *BOF* to *True*.
- Triggers an event handler for dataset scrolling events.
- Returns the actual number of records moved. In most cases, *Result* is the absolute value of *Distance*, but if *MoveBy* encounters the beginning-of-file or end-of-file before moving *Distance* records, *Result* will be less than the absolute value of *Distance*.

See also *First* method, *Last* method, *Next* method, *Prior* method

Next

Implements a generic method to position the cursor on the next record in the dataset.

```
procedure Next;
```

Description Do not call *Next* at the *TDataSet* level. In *TDataSet* descendents, call *Next* to position the cursor on the next record in the dataset and make it the current record. *Next* posts any changes to the current record and:

- Sets the *BOF* and *EOF* properties to *False*.
- Calls a **virtual**, **abstract** method to move to the next record, position the cursor on it, and make it the current record. For *TDataSet* this method is only declared, not implemented.
- Fetches any additional records required for display, such as those needed to fill out a grid control. In *TDataSet* this method calls another **virtual**, **abstract** method that is only declared, not implemented.
- Sets the *EOF* properties to *True* if the cursor was already on the last record in the dataset.
- Triggers an event handler for dataset scrolling events.

See also *First* method, *Last* method, *MoveBy* method, *Prior* method

Open

Opens the dataset.

```
procedure Open;
```

Description Call *Open* to set the *Active* property for the dataset to *True*. When *Active* is *True*, data can be read from and written to the database.

Setting *Active* to *True*:

- Triggers the *BeforeOpen* event handler if one is defined for the dataset.
- Sets the dataset state to *dsBrowse*.
- Opens a Borland Database Engine (BDE) cursor into the dataset, if appropriate (only applies to *TBDEDataSet* and its descendants, *TQuery*, *TStoredProc*, and *TTable*).
- Triggers the *After* Open event handler if one is defined for the dataset.

If an error occurs during the dataset open, dataset state is set to *dsInactive*, and the cursor is closed.

See also *DataSource* property, *State* property, *AfterOpen* event, *BeforeOpen* event, *Close* method

Post

Implements a virtual method to write a modified record to the database.

```
procedure Post; virtual;
```

Description *TDataSet* implements a virtual method to write a modified record to the database. Dataset methods that change the dataset state, such as *Edit*, *Insert*, or *Append*, or that move from one record to another, such as *First*, *Last*, *Next*, and *Prior* automatically call *Post*.

All descendents of *TDataSet* redeclare and implement *Post* to write a record to the database.

See also *State* property, *AfterPost* event, *BeforePost* event, *Append* method, *Edit* method, *First* method, *Insert* method, *Last* method, *Next* method, *Prior* method, *Resync* method, *UpdateRecord* method

Prior

Implements a generic method to position the cursor on the previous record in the dataset.

```
procedure Prior;
```

Description Do not call *Prior* at the *TDataSet* level. In *TDataSet* descendents, call *Prior* to position the cursor on the previous record in the dataset and make it the current record. *Prior* posts any changes to the current record and:

- Sets the *BOF* and *EOF* properties to *False*.
- Calls a **virtual, abstract** method to move to the previous record, position the cursor on it, and make it the current record. For *TDataSet* this method is only declared, not implemented.
- Fetches any additional records required for display, such as those needed to fill out a grid control. In *TDataSet* this method calls another **virtual, abstract** method that is only declared, not implemented.
- Sets the *BOF* properties to *True* if the cursor was already on the first record in the dataset.
- Triggers an event handler for dataset scrolling events.

See also *First* method, *Last* method, *MoveBy* method, *Prior* method

Refresh

Implements a generic method to refetch data from the database to update a dataset's view of data.

```
procedure Refresh;
```

Description Do not call *Refresh* for a *TDataSet* object. Instead call the redeclared and reimplemented *Refresh* method for a *TDataSet* descendent. *Refresh* ensures that an application has the latest data from a database. For example, when an application turns off filtering for a dataset, it should immediately call *Refresh* to display all records in the dataset, not just those that used to meet the filter condition.

See also *CheckBrowseMode* method, *Resync* method, *UpdateCursorPos* method

Resync

Refetches the current record and the records that precede and follow it.

```
type
  TResyncMode = set of (rmExact, rmCenter);

procedure Resync(Mode: TResyncMode); virtual;
```

Description *Resync* is an internal routine used by other dataset methods to resynchronize the Delphi dataset with underlying physical data when making calls that may change the cursor position. Applications should use the *Refresh* method instead of calling *Resync*.

Mode indicates optional processing that *Resync* should handle.

If *Mode* contains *rmExact*, *Resync* raises an exception if *Resync* is called when the cursor is not positioned on a valid record.

If *Mode* contains *rmCenter*, *Resync* positions the current record in the center of all buffered records.

Regardless of *Mode*, *Resync* also activates the buffers for the current record, retrieves prior and subsequent records for display purposes, and triggers a dataset change event.

See also *Refresh* method

SetFields

Sets the values for all fields in a record

```
procedure SetFields(const Values: array of const);
```

Description Call *SetFields* to set values for some or all fields in the current record at the same time.

Values contains the values to insert into each field. Values are assigned to the record based on the order of columns in the table or tables underlying the dataset. These values can be literals, variables, NULL, or **nil**. If *Values* contains fewer values than there are fields in the record, all records for which values are not provided are assigned a NULL value. A NULL value overwrites any existing value in such fields.

Before calling *SetFields*, call *Edit* to put the dataset into *dsEdit* state. After calling *SetFields*, call *Post* to write the changes to the database.

Note To set values for some fields while retaining existing values for others, pass **nil** for each field that should not change.

See also *AppendRecord* method, *InsertRecord* method

Translate

Implements a virtual procedure to copy a data string from a source to a destination, optionally performing character translation during the copy.

```
procedure Translate(Src, Dest: PChar; ToOem: Boolean); virtual;
```

Description This function copies a source string to a destination string, and ignores the value passed in *ToOem*. Ordinarily an application should not call *Translate* at the *TDataSet* level, but instead call a redeclared and implemented *Translate* in a descendent object such as *TTable* or a developer-derived object. *Translate* provides a fallback method for derived dataset objects that do not reimplement *Translate*.

UpdateCursorPos

Positions the cursor on the current record.

```
procedure UpdateCursorPos;
```

Description *UpdateCursorPos* is an internal routine used by many dataset methods to ensure that the physical cursor is positioned on the current record. Normally an application should not need to call *UpdateCursorPos*. Typically

UpdateCursorPos is called to ensure that the physical cursor position matches the logical cursor position.

See also *RecNo* property, *CursorPosChanged* method, *Refresh* method, *Resync* method

UpdateRecord

Triggers a data event for a record update.

```
procedure UpdateRecord;
```

Description *UpdateRecord* is an internal routine used by some dataset methods to trigger an *OnUpdateRecord* event if cached updates are enabled. Applications should not need to call *UpdateRecord* directly unless they provide custom dataset methods that bypass dataset methods.

See also *OnUpdateRecord* event

TDataSetTableProducer

Unit: DBWeb

Palette: Internet

TObject • TPersistent • TComponent • TCustomContentProducer • TDSTableProducer • TDataSetTableProducer

TDataSetTableProducer assembles a sequence of HTML commands to display the records from a *TDataSet* object in a tabular format.

Description Use *TDataSetTableProducer* to produce an HTML table for the records of a *TDataSet* object. If the dataset is a query and it requires input parameters from an HTTP request message, use *TQueryTableProducer* instead.

TDataSetTableProducer allows an application to create images of a dataset for an HTTP response message.

See also *TDataSet* object, *TQueryTableProducer* object, *TWebResponse* object

Properties

The following table lists the properties of *TDataSetTableProducer*:

Property	Visibility	Default	Documented in
Caption	published		TDSTableProducer
CaptionAlignment	published	caDefault	TDSTableProducer
Columns	published		TDSTableProducer
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent

Property	Visibility	Default	Documented in
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DataSet	published		TDataSetTableProducer
DesignInfo	public		TComponent
Dispatcher	published		TCustomContentProducer
Editor	public		TDSTableProducer
Footer	published		TDSTableProducer
Header	published		TDSTableProducer
MaxRows	published	20	TDSTableProducer
Name	published		TComponent
Owner	public, read-only		TComponent
RowAttributes	published		TDSTableProducer
TableAttributes	published		TDSTableProducer
Tag	published	0	TComponent
VCLComObject	public		TComponent

DataSet

DataSet is the *TDataSet* object where the dataset table producer looks for the records it displays in an HTML table.

property DataSet: TDataSet;

Description Set *DataSet* to specify the dataset that contains the records to be displayed in an HTML table. Use *DataSet* to set the dataset for the *TDataSetTableProducer* rather than the *DataSource* property. Dataset table producers do not use an external *TDataSource* object because they do not maintain record currency with other data objects in the application.

The individual field bindings specified by the *THTMLTableColumn* objects in the *Columns* property refer to fields in *DataSet*.

See also *Columns* property, *DataSource* property, *THTMLTableColumn.FieldName* property

Events

The following table lists the events of *TDataSetTableProducer*:

Event	Visibility	Documented in
OnCreateContent	published	TDSTableProducer
OnFormatCell	published	TDSTableProducer
OnGetTableCaption	published	TDSTableProducer

D

Methods

The following table lists the methods of *TDataSetTableProducer*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginUpdate	public	TDSTableProducer
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Content	public	TDataSetTableProducer
ContentFromStream	public	TCustomContentProducer
ContentFromString	public	TCustomContentProducer
Create	public	TDSTableProducer
DefaultHandler	public	TObject
Destroy	public	TDSTableProducer
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
EndUpdate	public	TDSTableProducer
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject

Method	Visibility	Documented in
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Content

Content returns the HTML representation of the records in the dataset.

```
function Content: string; override;
```

Description Call *Content* to retrieve the HTML representation specified by the properties of the *TDataSetTableProducer* object. *Content* calls the *OnCreateContent* event handler. If the *OnCreateContent* event handler indicates that the *Content* method should continue, *Content* returns the *Header* property, followed by an HTML table image of the records in the dataset, followed by the *Footer* property. If the *OnCreateContent* event handler indicates that the *Content* method should not continue, *Content* returns an empty string.

The string returned by *Content* can be used as the *Content* property of the response to an HTTP request message.

See also *Footer* property, *Header* property, *OnCreateContent* event, *TWebRequest* object, *TWebResponse.Content* property

TDataSetUpdateObject

Unit: DBTables

TObject • TPersistent • TComponent • TDataSetUpdateObject

Implements the base class for update objects used to update otherwise un-updatable queries when cached updates are enabled.

Description *TDataSetUpdateObject* is the base class for the *TUpdateSQL* object. It declared protected versions of one property and one method that are reimplemented in *TUpdateSQL*. Developers creating their own update objects may want to examine the source code for *TDataSetUpdateObject* to evaluate its functionality.

See also *TUpdateSQL* object

Properties

The following table lists the properties of *TDataSetUpdateObject*:

Property	Visibility	Default	Documented in
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent

D

Property	Visibility	Default	Documented in
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DesignInfo	public		TComponent
Name	published		TComponent
Owner	public, read-only		TComponent
Tag	published	0	TComponent
VCLComObject	public		TComponent

Methods

The following table lists the methods of *TDataSetUpdateObject*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TComponent
DefaultHandler	public	TObject
Destroy	public	TComponent
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject

Method	Visibility	Documented in
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

TDataSource

Unit: DB

Palette: Data Access

TObject • TPersistent • TComponent • TDataSource

TDataSource provides an interface between a dataset component and data-aware controls on a form.

Description

Use *TDataSource* to provide a conduit between a dataset and data-aware controls on a form that enable display, navigation, and editing of the data underlying the dataset.

All datasets must be associated with a data source component if their data is to be displayed and manipulated in data-aware controls. Similarly, each data-aware control needs to be associated with a data source component in order for the control to receive and manipulate data.

Data source components also link datasets in master-detail relationships.

See also *TDataSet* object

Properties

The following table lists the properties of *TDataSource*:

Property	Visibility	Default	Documented in
AutoEdit	published	True	TDataSource
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
DataSet	published		TDataSource

D

Property	Visibility	Default	Documented in
DesignInfo	public		TComponent
Enabled	published	True	TDataSource
Name	published		TComponent
Owner	public, read-only		TComponent
State	public, read-only		TDataSource
Tag	published	0	TComponent
VCLComObject	public		TComponent

AutoEdit

Determines if a data source component automatically calls a dataset's *Edit* method when a data-aware control associated with the data source receives focus.

property AutoEdit: Boolean;

Description *AutoEdit* is *True* by default. If *AutoEdit* is *True*, then when a user attempts to modify the data displayed by the control the underlying dataset's *Edit* method is called.

Set *AutoEdit* to *False* to protect data from unintentional modification. Even if *AutoEdit* is *False*, an application can call a dataset's *Edit* method directly to permit data modification.

See also *DataSet* property, *Enabled* property, *OnDataChange* event, *OnStateChange* event, *Edit* method, *TDataSet.Edit* method

DataSet

Specifies the dataset for which the data source component serves as a conduit to data-aware controls.

property DataSet: TDataSet;

Description Set *DataSet* to the name of an existing dataset component either at design time, or at runtime. By changing the value of *DataSet* at runtime an application can effectively use the same data-aware controls to display and edit data in different datasets.

Note To link a dataset that resides in a data module to a form at design time, choose File | Use unit.

See also *AutoEdit* property, *Enabled* property

Enabled

Determines if the data-aware controls associated with the data source component display data.

property Enabled: Boolean;

Description Use *Enabled* to control whether or not data-aware controls connected to a data source display data. If *Enabled* is *True* (the default), data are displayed. If *Enabled* is *False*, all controls associated with the data source are blank.

Note While an application can set *Enabled* to *False* to blank out data-aware controls during processing of multiple records, a better choice is to use the *TDataSet EnableControls* and *DisableControls* methods. Using these methods prevents the controls from blanking out during processing.

See also *State* property, *TDataSet.DisableControls* method, *TDataSet.EnableControls* method

State

Reads the current state of the dataset component associated with a data source component.

```
property State: TDataSetState;
```

Description Examine *State* to determine the current state of the dataset associated with a data source component. The state of a dataset indicates the action, if any, taking place against the dataset. The following table summarizes possible values for *State*:

State	Meaning
dsInactive	Dataset is closed, DataSet property is unassigned, or Enabled property is False.
dsBrowse	Dataset is open for viewing. This is the default state for a dataset.
dsEdit	Dataset is open. The current record can be modified.
dsInsert	Dataset is open. A new record is being inserted.
dsSetKey	Dataset is open. Searching for records based on indexed fields is enabled, or a SetRange operation is in progress. In either case a restricted set of data can be viewed. No data can be modified.
dsCalcFields	Dataset is open, and an OnCalcFields event is in progress. Fields that are not calculated cannot be modified.
dsFilter	Dataset is open, and a filter operation is in progress. A restricted set of data can be viewed. No data can be modified.

These values correspond to the values for a dataset's *State* property and are identical to them except that when *Enabled* is *False* or *DataSet* is unassigned, *State* is *dsInactive* regardless of the actual state of the dataset.

See also *DataSet* property, *Enabled* property, *Edit* method, *TDataSet.State* property, *TDataSet.OnCalcFields* event, *TDataSet.OnFilterRecord* event, *TDataSet.SetRange* method

D

Events

The following table lists the events of *TDataSource*:

Event	Visibility	Documented in
OnDataChange	published	TDataSource
OnStateChange	published	TDataSource
OnUpdateData	published	TDataSource

OnDataChange

OnDataChange occurs when the current record has been edited and the application moves from one field or record to another in a dataset associated with the data source component.

```
type
  TDataChangeEvent = procedure(Sender: TObject; Field: TField) of object;

property OnDataChange: TDataChangeEvent;
```

Description Write an *OnDataChange* event handler to take specific actions when a field in the current record has been edited and an application moves from one field or record to another in a dataset associated with a data source component. For example, methods that can trigger this event include the *Next* or *Prior* methods for the dataset.

Data-aware controls notify a data source of a data change because of:

- Scrolling to a new record.
- Modifications to a field's data.

The *Field* parameter is **nil** if more than one of the fields changed simultaneously, such as when moving from one record to another. Otherwise, *Field* indicates the edited field.

OnDataChange is especially useful in applications that must synchronize data display in controls that are not data aware.

See also *OnUpdateData* event, *OnStateChange* event

OnStateChange

OnStateChange occurs when the state of a data source component's dataset changes.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property OnStateChange: TNotifyEvent;
```

Description Write an *OnStateChange* event handler to take specific actions when the *State* property changes for a dataset associated with a data source component.

During the course of a normal connection to a database, a dataset's state changes frequently. For example, each time a user starts editing a field in a data-aware control the dataset's *State* property is changed from *dsBrowse* to *dsEdit* if the *State* is not already *dsEdit*. An *OnStateChange* event handler might respond to changes in state by taking actions such as disabling or enabling menu items or buttons.

See also *OnUpdateData* event, *TDataSet.State* property

OnUpdateData

OnUpdateData occurs when the data in the current record is about to be updated.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

property OnUpdateData: TNotifyEvent;
```

Description Write an *OnUpdateData* event handler to take specific actions prior to posting data changes to the database. If an application posts changes to the database and an *OnUpdateData* event handler exists, the handler is called before the dataset's *Post* method is executed. An *OnUpdateData* event handler might be useful for performing additional data processing or validation before posting.

See also *OnDataChange* event, *OnStateChange* event, *TDataSet.BeforePost* method

Methods

The following table lists the methods of *TDataSource*:

Method	Visibility	Documented in
Assign	public	TPersistent
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Create	public	TDataSource
DefaultHandler	public	TObject
Destroy	public	TDataSource
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
Edit	public	TDataSource
FieldAddress	public	TObject

D

Method	Visibility	Documented in
FindComponent	public	TComponent
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsLinkedTo	public	TDataSource
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RemoveComponent	public	TComponent
SafeCallException	public	TComponent

Create

Creates an instance of a *TDataSource* component.

constructor Create(AOwner: TComponent);

Description Call *Create* to instantiate a *TDataSource* component at runtime. For a data source component added to a data module at design time Delphi automatically inserts a call to *Create*.

Create instantiates a data link list, and sets the *Enabled* and *AutoEdit* properties of the component to *True*.

See also *AutoEdit* property, *Enabled* property, *Destroy* method

Destroy

Destroys the instance of a data source component.

destructor Destroy;

Description Do not call *Destroy* directly. Instead, an application should call *Free*, which verifies that the data source component is not already freed before calling *Destroy*.

Destroy sets *OnStateChange* to **nil**, sets the dataset for the component to **nil**, and frees the data link list before calling the inherited destructor for the component.

See also *DataSet* property, *OnStateChange* event, *Create* method, *TObject.Free* method

Edit

Determines whether or not to call the *Edit* method for a dataset associated with a data source component.

```
procedure Edit;
```

Description Call *Edit* to enable a data source component to control a user's ability to edit a dataset associated with the data source. *Edit* checks that the *AutoEdit* property of the data source is *True* and that its *State* property is *dsBrowse* before calling the dataset's *Edit* method. Checking these properties before calling the dataset's *Edit* method ensures that the dataset is in a state that supports editing, and that the data source allows editing as well.

See also *AutoEdit* property, *State* property, *TDataSet.Edit* method

IsLinkedTo

Checks that a data source is linked to a specified dataset.

```
function IsLinkedTo(DataSet: TDataSet): Boolean;
```

Description Call *IsLinkedTo* to verify that a data source is linked to a specific dataset. *DataSet* is the name of the dataset to which the data source should be linked.

If the data source already uses the specified dataset, *IsLinkedTo* returns *True*. If the data source does not use the specified dataset, but the dataset exists, then *IsLinkedTo* sets the *DataSet* property of the data source component to the one specified in the *DataSet* parameter, and returns *True*.

If the data source does not use the specified dataset, and the dataset does not already have a data source of its own, *IsLinkedTo* returns *False*.

Note *IsLinkedTo* is a low-level method mainly of interest to component developers who need to detect circular data references. Otherwise *IsLinkedTo* should not be called directly.

See also *DataSet* property, *TDataSet.IsLinkedTo* method

TDateField

Unit: DB

TObject • TPersistent • TComponent • TField • TDateTimeField • TDateField

A *TDateField* object represents a date field in a dataset.

Description *Date* fields contain date values.

TDateField introduces no new properties, events, or methods. It overrides the *DataSize* property and *Create* method to reflect the specifics of a field of type *ftDate*.

As a descendant of *TDateTimeField*, *TDateField* includes many properties, methods, and events that are useful for managing the value and properties of a temporal field in a database.

See also *TTimeField* component

Properties

The following table lists the properties of *TDateField*:

Property	Visibility	Default	Documented in
Alignment	published	taLeftJustify	TField
AsBoolean	public		TField
AsCurrency	public		TField
AsDateTime	public		TDateTimeField
AsFloat	public		TDateTimeField
AsInteger	public		TField
AsString	public		TDateTimeField
AsVariant	public		TDateTimeField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
CurValue	public, read-only		TField
CustomConstraint	published		TField
DataSet	public		TField
DataSize	public, read-only		TDateField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayFormat	published		TDateTimeField
DisplayLabel	published		TField
DisplayName	public, read-only		TField

Property	Visibility	Default	Documented in
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditMask	published		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	public		TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TDateTimeField
VCLComObject	public		TComponent
Visible	published	True	TField

DataSize

DataSize is the amount of memory that is needed to store the value of a date field in its native database format.

property DataSize: Word;

D

Description Check *DataSize* to determine the number of bytes required to store a date field component's value. Use *DataSize* to determine the size needed for a buffer when working with the *GetData* and *SetData* methods.

For date fields, *DataSize* is always 4. This is the size of date values in the database tables, not the size of the *TDateTime* data type used to store and manipulate date values by *TDateField*.

See also *GetData* method, *SetData* method, *Size* property, *Value* property

Events

The following table lists the events of *TDateField*:

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TDateField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TDateField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject

Method	Visibility	Documented in
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create creates an instance of *TDateField*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TDateField*. After calling the inherited constructor, *Create* sets the *DataType* to *ftDate*.

It is seldom necessary to call *Create* directly because a date field object is instantiated automatically for all date fields in a dataset.

See also *DataType* property, *TField.Create* method

D

TDateTimeField

Unit: DB

TObject • TPersistent • TComponent • TField • TDateTimeField

A *TDateTimeField* object represents a date-time field in a dataset.

Description Date-time fields contain combined date and time information, sometimes called timestamps.

TDateTimeField introduces new properties to convert between date-time values and other data types, and to format the value of a date-time field for display. As a descendant of *TField*, *TDateTimeField* includes many properties, methods, and events that are useful for managing the value and properties of a field in a database.

TDateTimeField is also the direct ancestor of two other temporal field types: *TDateField* and *TTimeField*.

See also *TDateField* object, *TTimeField* object

Properties

The following table lists the properties of *TDateTimeField*:

Property	Visibility	Default	Documented in
Alignment	published	taLeftJustify	TField
AsBoolean	public		TField
AsCurrency	public		TField
AsDateTime	public		TDateTimeField
AsFloat	public		TDateTimeField
AsInteger	public		TField
AsString	public		TDateTimeField
AsVariant	public		TDateTimeField
AttributeSet	public		TField
Calculated	public	False	TField
CanModify	public, read-only		TField
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ConstraintErrorMessage	published		TField
CurValue	public, read-only		TField
CustomConstraint	published		TField

Property	Visibility	Default	Documented in
DataSet	public		TField
DataSize	public, read-only		TDateTimeField
DataType	public, read-only		TField
DefaultExpression	published		TField
DesignInfo	public		TComponent
DisplayFormat	published		TDateTimeField
DisplayLabel	published		TField
DisplayName	public, read-only		TField
DisplayText	public, read-only		TField
DisplayWidth	published		TField
EditMask	published		TField
EditMaskPtr	public, read-only		TField
FieldKind	published	fkData	TField
FieldName	published		TField
FieldNo	public, read-only		TField
ImportedConstraint	published		TField
Index	published		TField
IsIndexField	public, read-only		TField
IsNull	public, read-only		TField
KeyFields	published		TField
Lookup	public		TField
LookupCache	published	False	TField
LookupDataSet	published		TField
LookupKeyFields	published		TField
LookupList	public, read-only		TField
LookupResultField	published		TField
Name	published		TComponent
NewValue	public		TField
Offset	public, read-only		TField
OldValue	public, read-only		TField
Origin	published		TField
Owner	public, read-only		TComponent
ReadOnly	published	False	TField
Required	published	False	TField
Size	public		TField
Tag	published	0	TComponent
Text	public		TField
ValidChars	public		TField
Value	public		TDateTimeField
VCLComObject	public		TComponent
Visible	published	True	TField

AsDateTime

AsDateTime converts the value of the date-time field to or from a date-time value.

```
property AsDateTime: TDateTime;
```

Description Use *AsDateTime* to

- Assign a date-time value to a date-time field.
- Assign the date-time field value to a date-time variable.

The time portion of the date-time value is ignored when setting the *AsDateTime* property on a field where the *DataType* is *ftDate*. The date portion of the date-time value is ignored when setting the *AsDateTime* property on a field where the *DataType* is *ftTime*.

Similarly, reading the *AsDateTime* property of a field with *DataType* set to *ftDate* yields a date-time value of 12:00 a.m. on the date value of the field. Reading the *AsDateTime* property of a field with *DataType* set to *ftTime* yields a date-time value of the time value of the field on Dec. 30, 1899.

Reading or setting the *AsDateTime* property for a field with *DataType* set to *ftDateTime* treats both the date and time portions of the date-time value as meaningful.

AsDateTime uses date-time values stored in the *TDateTime* type. The integral part of a *TDateTime* value is the number of days that have passed since Dec. 30 1899. The fractional part of a *TDateTime* value is the fraction of a 24 hour day that has elapsed.

Following are some examples of *TDateTime* values and their corresponding dates and times:

Value	Date and time
0	Dec. 30 1899 12:00 a.m.
2.75	Jan. 1 1900 6:00 p.m.
–1.25	Dec. 29 1899 6:00 a.m.
35065	Jan. 1 1996 12:00 a.m.

To find the number of days between two dates, simply subtract the two values. Likewise, to increment a date and time value by a certain fractional number of days, simply add the fractional number to the date and time value.

Note Because *TDateTimeField* stores date and time values in the *TDateTime* type, *AsDateTime* is the same as *Value* for date-time fields. Use *AsDateTime* when working with field components where the actual class of the object is not known.

See also *DataType* property, *Value* property

AsFloat

AsFloat converts the value of the date-time field to or from a floating-point value.

property AsFloat: Double;

Description Use *AsFloat* to

- Assign a floating-point value to a date-time field.
- Assign the date-time field value to a floating-point variable.

For example, this code assigns a floating-point value to date-time field value:

```
MyDateTimeField.AsFloat := 35065.75;  { A date and time of of Jan 1, 1996
   6:00 p.m. }
```

To assign date-time field value to a floating-point value, use code similar to this:

```
FloatVariable := MyDateTimeField.AsFloat;
```

TDateTimeField converts dates to the number of days that have passed since Dec. 30 1899. Time values are converted to the fraction of a 24-hour day that has elapsed. The floating-point value for a date-time quantity is the sum of the date and time values.

The fractional (time) portion of the floating-point value is ignored when setting the *AsFloat* property on a field where the *DataType* is *ftDate*. The integral (date) portion of the floating-point value is ignored when setting the *AsFloat* property on a field where the *DataType* is *ftTime*.

Similarly, reading the *AsFloat* property of a field with *DataType* set to *ftDate* yields a floating-point value with no fractional (time) part. Reading the *AsFloat* property of a field with *DataType* set to *ftTime* yields a floating-point value greater than –1 and less than 1.

See also *DataType* property, *AsDateTime* property

AsString

AsString converts the value of the date-time field to or from a string.

property AsString: **string**;

Description Use *AsString* to

- Assign a string to a date-time field.
- Assign the date-time field value to a string.

For example, this code assigns a string to a date-time field value:

```
MyDateTimeField.AsString := '1/1/1997 8:00 am';
```

To assign date-time field value to a floating-point variable, use code similar to this:

```
StringVariable := MyDateTimeField.AsString;
```

D

If the *DataType* property value of the date-time field is *ftDate*, setting the *AsString* property uses the *StrToDate* function to convert the string to a date value. If the *DataType* is *ftTime*, setting the *AsString* property uses the *StrToTime* function. If the *DataType* is *ftDateTime*, setting the *AsString* property uses the *StrToDateTime* function.

If the *DisplayFormat* property is set, reading the *AsString* property yields the value of the date-time field formatted according to the *DisplayFormat*. If *DisplayFormat* has not been set, reading the *AsString* property yields the value of the date-time field formatted using the date and time format specified by the Windows Control Panel.

See also *DataType* property, *DisplayFormat property*, *StrToDate function*, *StrToDateTime* function, *StrToTime* function

AsVariant

AsVariant converts the value of a date-time field to or from a variant value.

property AsVariant: Variant;

Description Use *AsVariant* to

- Assign the date-time field value to variant variable.
- Assign a variant value to the date-time field.

Using variants is useful when the actual field type and the data type are unknown.

See also *AsDateTime* property, *AsFloat* property, *AsString* property

DataSize

DataSize is the amount of memory that is needed to store the value of a date-time field in its native database format.

property DataSize: Word;

Description Check *DataSize* to determine the number of bytes required to store a date-time field component's value. Use *DataSize* to determine the size needed for a buffer when working with the *GetData* and *SetData* methods.

For date-time fields, *DataSize* is always 8.

See also *GetData* method, *SetData* method

DisplayFormat

DisplayFormat contains the string that overrides the automatic formatting of the date-time field for display purposes.

property DisplayFormat: **string**;

Description Use *DisplayFormat* to specify a formatting string that is used to format the value in a date-time field when the field's value appears as a string.

If *DisplayFormat* is not assigned a string, the value is formatted according to the default specified by the Windows Control Panel.

Construct a *DisplayFormat* string using these format specifiers:

Specifier	Displays
c	The date using the format given by the *ShortDateFormat* global variable, followed by the time using the format given by the *LongTimeFormat* global variable. The time is not displayed if the fractional part of the DateTime value is zero.
d	The day as a number without a leading zero (1–31).
dd	The day as a number with a leading zero (01–31).
ddd	The day as an abbreviation (Sun–Sat) using the strings given by the *ShortDayNames* global variable.
dddd	The day as a full name (Sunday–Saturday) using the strings given by the *LongDayNames* global variable.
ddddd	The date using the format given by the *ShortDateFormat* global variable.
dddddd	The date using the format given by the *LongDateFormat* global variable.
m	The month as a number without a leading zero (1–12). If the *m* specifier immediately follows an *h* or *hh* specifier, the minute rather than the month is displayed.
mm	The month as a number with a leading zero (01–12). If the *mm* specifier immediately follows an *h* or *hh* specifier, the minute rather than the month is displayed.
mmm	The month as an abbreviation (Jan–Dec) using the strings given by the *ShortMonthNames* global variable.
mmmm	The month as a full name (January–December) using the strings given by the *LongMonthNames* global variable.
yy	The year as a two-digit number (00–99).
yyyy	The year as a four-digit number (0000–9999).
h	The hour without a leading zero (0–23).
hh	The hour with a leading zero (00–23).
n	The minute without a leading zero (0–59).
nn	The minute with a leading zero (00–59).
s	The second without a leading zero (0–59).
ss	The second with a leading zero (00–59).
t	The time using the format given by the *ShortTimeFormat* global variable.
tt	The time using the format given by the *LongTimeFormat* global variable.
am/pm	The time using the 12-hour clock for the preceding *h* or *hh* specifier, followed by "am" for any hour before noon, or "pm" for any hour after noon. The *am/pm* specifier can use lower, upper, or mixed case, and the result is displayed accordingly.

D

Specifier	Displays
a/p	The time using the 12-hour clock for the preceding *h* or *hh* specifier, followed by "a" for any hour before noon, or "p" for any hour after noon. The *a/p* specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
ampm	The time using the 12-hour clock for the preceding *h* or *hh* specifier, followed by the contents of the *TimeAMString* global variable for any hour before noon, or the contents of the *TimePMString* global variable for any hour after noon.
/	The date separator character given by the *DateSeparator* global variable.
:	The time separator character given by the *TimeSeparator* global variable.
'xx'/"xx"	Characters enclosed in single or double quotes are displayed as-is, with no formatting changes.

Format specifiers may be written in uppercase or lowercase letters; both produce the same result.

If the string given by the *Format* parameter is empty, the date and time value is formatted as if a *c* format specifier had been given.

See also *AsString* property

Value

Value is the value of the data in the date-time field.

```
property Value: TDateTime;
```

Description Use *Value* to read data directly from and write data directly to a date-time field component at runtime. *TDateTimeField* components store and manipulate their data as *TDateTime* values. Thus, for date-time fields, the *Value* property is the same as the *AsDateTime* property.

See also *AsDateTime* property

Events

The following table lists the events of *TDateTimeField*:

Event	Visibility	Documented in
OnChange	published	TField
OnGetText	published	TField
OnSetText	published	TField
OnValidate	published	TField

Methods

The following table lists the methods of *TDateTimeField*:

Method	Visibility	Documented in
Assign	public	TField
AssignValue	public	TField
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TField
Create	public	TDateTimeField
DefaultHandler	public	TObject
Destroy	public	TField
DestroyComponents	public	TComponent
Destroying	public	TComponent
Dispatch	public	TObject
FieldAddress	public	TObject
FindComponent	public	TComponent
FocusControl	public	TField
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetData	public	TField
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InstanceSize	public	TObject
IsBlob	public	TField
IsValidChar	public	TField
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject

Method	Visibility	Documented in
RefreshLookupList	public	TField
RemoveComponent	public	TComponent
SafeCallException	public	TComponent
SetData	public	TField
SetFieldType	public	TField
Validate	public	TField

Create

Create instantiates a *TDateTimeField* object.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize an instance of *TDateTimeField*. After calling the inherited constructor, *Create* sets the *DataType* to *ftDateTime*.

It is seldom necessary to call *Create* directly, because a date-time field object is instantiated automatically for all date-time fields in a dataset.

See also *DataType* property, *TField.Create* method

TDateTimePicker

Unit: ComCtrls Palette: Win32

TObject • TPersistent • TComponent • TControl • TWinControl • TDateTimePicker

TDateTimePicker displays a list box for entering dates or times.

Description *TDateTimePicker* is a visual component designed specifically for entering dates or times. In *dmComboBox* date mode, it resembles a list box or combo box, except that the drop-down list is replaced with a calendar illustration; users can select a date from the calendar. Dates or times can also be selected by scrolling with Up and Down arrows and by typing.

See also *TComboBox* object, *TListBox* object, *DateMode* property

Properties

The following table lists the properties of *TDateTimePicker*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
CalAlignment	published		TDateTimePicker

Property	Visibility	Default	Documented in
CalColors	published		TDateTimePicker
Checked	published	True	TDateTimePicker
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
Date	published		TDateTimePicker
DateFormat	published		TDateTimePicker
DateMode	published		TDateTimePicker
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
Kind	published		TDateTimePicker
Left	published		TControl
MaxDate	published		TDateTimePicker
MinDate	published		TDateTimePicker
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	True	TControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl

Property	Visibility	Default	Documented in
ParentWindow	public		TWinControl
ParseInput	published		TDateTimePicker
PopupMenu	published		TControl
ShowCheckbox	published	False	TDateTimePicker
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Time	published		TDateTimePicker
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

CalAlignment

Determines the alignment of the drop-down calendar.

```
type TDTCalAlignment = (dtaLeft, dtaRight);
property CalAlignment: TDTCalAlignment;
```

Description Set *CalAlignment* to *dtaLeft* or *dtaRight* to align the drop-down calendar on the left or right side of the component. This property is applicable only if *Kind* = *dtkDate* and *DateMode* = *dmComboBox*, and makes little difference except when the *TDateTimePicker* component is larger than the drop-down calendar.

See also *DateMode* property, *Kind* property

CalColors

Determines the colors of the drop-down calendar.

```
property CalColors: TDateTimeColors;
```

Description The *CalColors* property points to a *TDateTimeColors* object that defines colors for the drop-down calendar. The easiest way to select colors for the calendar is to edit *CalColors* in the Object Inspector. This property is applicable only when *Kind* = *dtkDate* and *DateMode* = *dmComboBox*.

See also *DateMode* property, *Kind* property, *TDateTimeColors* object

Checked

Indicates whether the check box next to the date or time is selected.

```
property Checked: Boolean default True;
```

Description *Checked* returns *True* if the check box in the *TDateTimePicker* component is selected. Setting *Checked* to *True* at design time makes the check box appear selected by default.

See also *ShowCheckbox* property

Date

The date entered by the user.

```
type TDate = type TDateTime;
```

```
property Date: TDate;
```

Description *Date* represents the date entered by the user. Set *Date* in program code or in the Object Inspector to determine which date is selected by default. (Applies only when *Kind* = *dtkDate*.)

See also *Kind* property, *TDateTime* type, *Time* property

DateFormat

The format in which the date is presented.

```
type TDTDateFormat = (dfShort, dfLong);
```

```
property DateFormat: TDTDateFormat;
```

Description If *DateFormat* is set to *dfShort*, the date appears like this: 3/21/97. If *DateFormat* is set to *dfLong*, the date appears like this: Friday, March 21, 1997. (Applies only when *Kind* = *dtkDate*.)

See also *Kind* property

DateMode

Determines the method of date selection used by the component.

```
type TDTDateMode = (dmComboBox, dmUpDown);
```

```
property DateMode: TDTDateMode;
```

Description If *DateMode* is *dmComboBox*, the component has a drop-down calendar illustration from which the user can select a date. If *DateMode* is *dmUpDown*, the component has Up/Down arrows with which the user can adjust the date. (Applies only when *Kind* = *dtkDate*.)

See also *Kind* property

Kind

Determines whether the component is a date selector or a time selector.

```
type TDateTimeKind = (dtkDate, dtkTime);
property Kind: TDateTimeKind;
```

Description If *Kind* is *dtkDate*, the component is used to select a date. If *Kind* is *dtkTime*, the component is used to select a time.

See also *DateFormat* property, *DateMode* property

MaxDate

The latest allowable date that can be entered.

```
type TDate = type TDateTime;
property MaxDate: TDate;
```

Description Assign a value to *MaxDate* to limit the range of dates that the user can enter. Out-of-range dates raise the *EDateTimeError* exception. (Applies only when *Kind* = *dtkDate*.)

See also *EDateTimeError* object, *Kind* property, *MinDate* property, *TDateTime* type

MinDate

The earliest allowable date that can be entered.

```
type TDate = type TDateTime;
property MinDate: TDate;
```

Description Assign a value to *MinDate* to limit the range of dates that the user can enter. Out-of-range dates raise the *EDateTimeError* exception. (Applies only when *Kind* = *dtkDate*.)

See also *EDateTimeError* object, *Kind* property, *MaxDate* property, *TDateTime* type

ParseInput

Enables the *OnUserInput* event.

```
property ParseInput: Boolean;
```

Description If *ParseInput* is *True*, the *OnUserInput* event occurs whenever the user types into the component's edit box.

See also *OnUserInput* event

ShowCheckbox

Displays a check box next to the date or time.

```
property ShowCheckbox: Boolean;
```

Description Set *ShowCheckbox* to *True* to display a check box in the *TDateTimePicker* component.

See also *Checked* property

Time

The time entered by the user.

```
type TTime = type TDateTime;
```

```
property Time: TTime;
```

Description *Time* represents the time entered by the user. Set *Time* in program code or in the Object Inspector to determine which time is selected by default. (Applies only when *Kind* = *dtkTime*.)

See also *Date* property, *Kind* property, *TDateTime* type

Events

The following table lists the events of *TDateTimePicker*:

Event	Visibility	Documented in
OnChange	published	TDateTimePicker
OnClick	published	TControl
OnCloseUp	published	TDateTimePicker
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnDropDown	published	TDateTimePicker
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnStartDrag	published	TControl
OnUserInput	published	TDateTimePicker

OnChange

OnChange occurs when a date or time is entered.

```
property OnChange: TNotifyEvent;
```

Description Write an *OnChange* event handler to take specific action whenever the date or time entered in the component changes.

See also *TNotifyEvent* type

OnCloseUp

OnCloseUp occurs when the drop-down calendar closes.

```
property OnCloseUp: TNotifyEvent;
```

Description Write an *OnCloseUp* event handler to implement special processing that needs to occur only when the drop-down calendar closes.

See also *Kind* property, *DateMode* property, *OnDropDown* event, *TNotifyEvent* type

OnDropDown

OnDropDown occurs when the user opens the drop-down calendar by clicking the arrow at the right of the control.

```
property OnDropDown: TNotifyEvent;
```

Description Write an *OnDropDown* event handler to implement special processing that needs to occur only when the drop-down calendar is opened.

See also *Kind* property, *DateMode* property, *OnCloseUp* event, *TNotifyEvent* type

OnUserInput

OnUserInput occurs when the user types directly into the component's edit box.

```
type TDTParseInputEvent = procedure(Sender: TObject; const UserString: string;
  var DateAndTime: TDateTime; var AllowChange: Boolean) of object;

property OnUserInput: TDTParseInputEvent;
```

Description This event occurs only if *ParseInput* is set to *True*. Write an *OnUserInput* event handler implement special processing that needs to occur when the user types directly into the *TDateTimePicker* (rather than selecting with the drop-down calendar or scroll arrows). *UserString* is the string that the user types, and *DateAndTime* represents the value of the *Date* or *Time* property.

See also *Date* property, *ParseInput* property, *TDateTime* type, *Time* property

Methods

The following table lists the methods of *TDateTimePicker*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl

Method	Visibility	Documented in
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDateTimePicker
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDateTimePicker
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl

Method	Visibility	Documented in
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Creates and initializes a *TDateTimePicker* instance.

constructor Create(AOwner: TComponent); **override**;

Description *Create* calls the inherited constructor method and initializes the *Date* and *Time* properties to the current date and time. It also sets the following *ControlStyle* flags: *csCaptureMouse, csOpaque, csClickEvents, csDoubleClicks,* and *csFixedHeight*.

See also *ControlStyle* property, *Date* property, *Time* property, *TWinControl.Create* method

Destroy

Destroys the *TDateTimePicker* instance and frees its memory.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the control is not already destroyed, and only then calls *Destroy*.

See also *Free* method

TDBCheckBox

Unit: DBCtrls — Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TButtonControl • TCustomCheckBox • TDBCheckBox

TDBCheckBox is a data aware control that allows the user to select or deselect a single value.

Description Use *TDBCheckBox* to place a data-aware check box on a form. A check box presents an option to the user; the user can check it to select the option, or uncheck it to deselect the option. A database check box (*TDBCheckBox*) is much like an ordinary check box (*TCheckBox*), except that it is aware of the data in a particular field of a dataset. Because check boxes can represent only two values (checked and unchecked), database check boxes are most appropriate for boolean fields. They can, however, be used to group the values of any field into two sets.

For applications that don't require the data-aware capabilities of *TDBCheckBox*, use *TCheckBox* instead to conserve system resources.

See also *TBooleanField* object, *TCheckBox* object

Properties

The following table lists the properties of *TDBCheckBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
Alignment	published	taRightJustify	TCustomCheckBox
AllowGrayed	published	False	TCustomCheckBox
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Caption	published		TControl
Checked	public		TCustomCheckBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl

D

Property	Visibility	Default	Documented in
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DataField	published		TDBCheckBox
DataSource	published		TDBCheckBox
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Field	public, read-only		TDBCheckBox
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	True	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ReadOnly	published	False	TDBCheckBox
ShowHint	published		TControl
Showing	public, read-only		TWinControl
State	public	cbUnchecked	TCustomCheckBox
TabOrder	published	–1	TWinControl

Property	Visibility	Default	Documented in
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
ValueChecked	published		TDBCheckBox
ValueUnchecked	published		TDBCheckBox
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

DataField

DataField identifies the field from which the data-aware control displays data.

```
property DataField: string;
```

Description Set *DataField* to the field name of the field component that the check box represents. The dataset the field is located in is specified by the *DataSource* property.

See also *DataSource* property, *Field* property

DataSource

DataSource links the check box to a dataset.

```
property DataSource: TDataSource;
```

Description Specify the data source component that identifies the dataset the data is found in. To allow the check box to represent the data for a field, both the *DataSource* and the *DataField* properties must be set.

See also *DataField* property, *Field* property

Field

Field is the *TField* object whose current value the check box represents.

```
property Field: TField;
```

Description Use the *Field* object when you want to change the value of the data in the field programmatically.

See also *DataField* property, *DataSource* property, *TField* object

D

ReadOnly

ReadOnly determines whether the user can change the value of the field.

property ReadOnly: Boolean;

Description Use ReadOnly to specify whether the check box allows the user to change the field value by checking or unchecking the box. When *ReadOnly* is *True*, the checkbox can only be used to display the value of the field on the current record. When *ReadOnly* is *False*, the user can use the check box to change the field value.

See also *Field* property

ValueChecked

ValueChecked is the field value that corresponds to the checked state of the checkbox.

property ValueChecked: **string**;

Description Use *ValueChecked* to specify the field value represented by the check box when it is checked. If the value of the *ValueChecked* property is equal to the data in the field of the current record of the dataset, the database check box appears checked. When the user checks the database check box, the field value is set to *ValueChecked*.

ValueChecked can represent more than one value in a semicolon-delimited list of items. If any of the items matches the contents of the field of the current record in the dataset, the check box appears checked. For example, set the value of *ValueChecked* string like this:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

If the contents of the associated field is the string True, Yes, or On, the check box is checked. The value of the field is compared to *ValueChecked* in a case-insensitive comparison. If the user selects a check box where *ValueChecked* represents more than one value, the first item in the list is assigned to the field.

If the contents of the field of the current record matches a string specified as the value of the *ValueUnchecked* property, the check box appears unchecked. If the contents of the field matches no string in either *ValueChecked* or *ValueUnchecked*, the check box appears gray.

Note If the *DataField* of the database check box is a logical field, the check box is always checked if the contents of the field is *True*, and it is always unchecked if the contents of the field is *False*. The values of the *ValueChecked* and *ValueUnchecked* properties have no affect on logical fields.

See also *TBooleanField* object, *ValueUnchecked* property

ValueUnchecked

ValueUnchecked is the field value that corresponds to the unchecked state of the checkbox.

property ValueUnchecked: **string**;

Description Use *ValueUnchecked* to specify the field value represented by the check box when it is unchecked. If the value of the *ValueUnchecked* property is equal to the data in the field of the current record of the dataset, the database check box appears unchecked. When the user unchecks the database check box, the field value is set to *ValueUnchecked*.

ValueUnchecked can represent more than one value in a semicolon-delimited list of items. If any of the items matches the contents of the field of the current record in the dataset, the check box appears unchecked. For example, set the value of *ValueUnchecked* string like this:

```
DBCheckBox1.ValueUnchecked := 'False;No;Off';
```

If the contents of the associated field is the string False, No, or Off, the check box appears unchecked. The value of the field is compared to *ValueUnchecked* in a case-insensitive comparison. If the user unchecks a check box where *ValueUnchecked* represents more than one value, the first item in the list is assigned to the field.

If the contents of the field of the current record matches a string specified as the value of the *ValueChecked* property, the check box appears checked. If the contents of the field matches no string in either *ValueChecked* or *ValueUnchecked*, the check box appears gray.

Note If the *DataField* of the database check box is a logical field, the check box is always unchecked if the contents of the field is *False*, and it is always checked if the contents of the field is *True*. The values of the *ValueChecked* and *ValueUnchecked* properties have no affect on logical fields.

See also *ValueChecked* property

Events

The following table lists the events of *TDBCheckBox*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TDBCheckBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDBCheckBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDBCheckBox
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent

Method	Visibility	Documented in
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TDBCheckBox*.

constructor Create(AOwner: TComponent); **override**;

D

Description Call *Create* to instantiate a *TDBCheckBox* object at runtime. For *TDBCheckBox* objects placed on forms or data modules at design time, *Create* is called automatically.

After calling the inherited constructor to allocate and initialize a generic checkbox, *Create*

- Adds *csReplicatable* to *ControlStyle*.
- Creates and initializes the helper objects needed to link to a dataset.

See also *TCustomCheckBox.Create* method, *TFieldDataLink* object

Destroy

Destroy destroys an instance of *TDBCheckBox*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TDBCheckBox* object is not already freed and only then calls *Destroy*.

Destroy frees the helper objects of the *TDBCheckBox* before calling the inherited destructor.

See also *Free* method

TDBComboBox

Unit: DBCtrls Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomComboBox • TDBComboBox

TDBComboBox is a data-aware combo box control.

Description Use *TDBComboBox* to allow the user to change the value of a field on the current record in a dataset either by selecting an item from a list or by typing in the edit box part of the control. The selected item or entered text becomes the new value of the field if the database combo box's *ReadOnly* property is *False*. The combo box can be customized to enable or disable typing in the edit region of the control, to display the list as a drop down or as a permanently displayed list, to sort the items in the list, and so on.

See also *TDBListBox* object, *TDBLookupComboBox* object

Properties

The following table lists the properties of *TDBComboBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomComboBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DataField	published		TDBComboBox
DataSource	published		TDBComboBox
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
DropDownCount	published	8	TCustomComboBox
DroppedDown	public		TCustomComboBox
Enabled	published	True	TControl
Field	public, read-only		TDBComboBox
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
ItemHeight	published		TCustomComboBox
ItemIndex	public		TCustomComboBox
Items	published		TDBComboBox
Left	published		TControl
Name	published		TControl

Property	Visibility	Default	Documented in
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ReadOnly	published	False	TDBComboBox
SelLength	public		TCustomComboBox
SelStart	public		TCustomComboBox
SelText	public		TCustomComboBox
ShowHint	published		TControl
Showing	public, read-only		TWinControl
Sorted	published	False	TCustomComboBox
Style	published	csDropDown	TDBComboBox
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	public		TControl
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

DataField

DataField identifies the field from which the combo box displays data.

property DataField: **string**;

Description Set *DataField* to specify which field is represented by the combo box. The combo box displays the current value of this field, and allows the user to set the value of this field on the current record. The dataset the field is located in is specified by the *DataSource* property.

See also *DataSource* property, *Field* property

DataSource

DataSource links the combo box to a dataset.

property DataSource: TDataSource;

Description Specify the data source component that identifies the dataset the data is found in. To allow the combo box to represent the data for a field, both the *DataSource* and the *DataField* properties must be set.

See also *DataField* property, *TDataSource* object

Field

Field is the *TField* object whose current value the combo box represents.

```
property Field: TField;
```

Description Use the *Field* object to change the value of the data in the field programmatically.

See also *TField* object, *DataField* property

Items

Items contains the strings that appear in the list of the database combo box.

```
property Items: TStrings;
```

Description Use *Items* to specify the values the user can choose from. Because *Items* is an object of type *TStrings*, you can add, delete, insert, and move items using the *Add*, *Delete*, *Insert*, *Exchange*, and *Move* methods of the *TStrings* object. For example, to add each string to the combo box list, write a line of code such as the following:

```
DBComboBox1.Items.Add('AFieldValue');
```

The *ItemIndex* property determines which item is selected, if any.

See also *ItemIndex* property, *Selected* property

ReadOnly

ReadOnly determines if the user can change the value of the field.

```
property ReadOnly: Boolean;
```

Description Set *ReadOnly* to specify whether the combo box should be used for display purposes only. If *ReadOnly* is *True*, the combo box can only be used to display the value of the field on the current record. If *ReadOnly* is *False*, the user can use the combo box to edit the field's value.

See also *Field* property, *TDataSet.Edit* method

Style

Style determines how the combo box displays its items.

```
type TComboBoxStyle = (csDropDown, csSimple, csDropDownList, csOwnerDrawFixed,
  csOwnerDrawVariable);
```

property Style: TComboBoxStyle;

Description Use *Style* to specify whether the user can type values into the edit region as well as select values from the list, whether the list drops down when the user clicks a button or if it is always displayed, and so on. By default, *Style* is *csDropDown*, meaning that the combo box displays each item as a string in a drop-down list, and that users can edit the value in the edit region. *Style* can be any of the following values:

Value	Meaning
csDropDown	Creates a drop-down list with an edit box in which the user can enter values. All items are strings that represent the contents of the field on the current record, each item having the same height.
csSimple	Creates an edit box with a fixed list (list box) underneath. The length of the list is determined by the *Height* of the combo box. The user can change the contents of the field by typing in a new value or selecting a value from the list.
csDropDownList	Creates a drop-down list with an attached static text box. All items are the same height.The edit box is blank unless the current contents of the field matches one of the specified *Items* in the drop-down list. The user can change the contents of the field only by selecting one of the strings in the drop-down list.
csOwnerDrawFixed	Same as *csDropDownList* except that items are not drawn automatically. Items have the height specified by the *ItemHeight* property.
csOwnerDrawVariable	Same as *csDropDownList* except that items are not drawn automatically. Items can vary in height.

Owner-draw combo boxes can display items other than strings. For example, a combo box could display graphical images along with or instead of its strings. Owner-draw combo boxes require more programming, however, as the application must render the image for each item in the list.

When *Style* is *csOwnerDrawFixed* or *csOwnerDrawVariable*, the *OnDrawItem* event occurs when the list items need to be drawn. If *Style* is *csOwnerDrawVariable*, an *OnMeasureItem* event occurs as well.

See also *Height* property, *ItemHeight* property, *Items* property, *OnDrawItem* event, *OnMeasureItem* event

Events

The following table lists the events of *TDBComboBox*:

Event	Visibility	Documented in
OnChange	published	TCustomComboBox
OnClick	published	TControl
OnDblClick	published	TControl

Event	Visibility	Documented in
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnDrawItem	published	TCustomComboBox
OnDropDown	published	TCustomComboBox
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMeasureItem	published	TCustomComboBox
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TDBComboBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomComboBox
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDBComboBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDBComboBox
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject

Method	Visibility	Documented in
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl

Method	Visibility	Documented in
ScrollBy	public	TWinControl
SelectAll	public	TCustomComboBox
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TDBComboBox*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a *TDBComboBox* object at runtime. For *TDBComboBox* objects placed on forms or data modules at design time, *Create* is called automatically.

After calling the inherited constructor to allocate and initialize a generic combo box object, *Create* creates and initializes the helper objects needed to link to a dataset.

See also *TCustomComboBox.Create* method, *TFieldDataLink* object

Destroy

Destroy destroys an instance of *TDBComboBox*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TDBComboBox* object is not already freed and only then calls *Destroy*.

Destroy frees the helper objects of the *TDBComboBox* before calling the inherited destructor.

See also *Create* method, *Free* method

TDBCtrlGrid

Unit: DBCGrids — Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TDBCtrlGrid

TDBCtrlGrid displays records from a data source in a free-form layout.

Description Put a *TDBCtrlGrid* object on a form to display and edit the records from a database table or query. Unlike the *TDBGrid* object, which displays each record in a single row, *TDBCtrlGrid* allows a free-form layout for each record in the dataset. Each record is displayed in its own panel; Design one panel at design time and *TDBCtrlGrid* replicates that panel for each record displayed.

At runtime, users can use the database navigator (*TDBNavigator*) to move through data in the *TDBCtrlGrid* object, and to insert, delete, and edit the data. Edits made using the data grid are not posted to the underlying dataset until the user moves to a different record or closes the application.

See also *TDBCtrlPanel* object, *TDBGrid* object, *TDBNavigator* object

Properties

The following table lists the properties for *TDBCtrlGrid*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
AllowDelete	published	True	TDBCtrlGrid
AllowInsert	published	True	TDBCtrlGrid
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TDBCtrlGrid
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ColCount	published		TDBCtrlGrid
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DataSource	published		TDBCtrlGrid
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl

Property	Visibility	Default	Documented in
EditMode	public		TDBCtrlGrid
Enabled	published	True	TControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Orientation	published	goVertical	TDBCtrlGrid
Owner	public, read-only		TComponent
PanelBorder	published	gbRaised	TDBCtrlGrid
PanelCount	public, read-only		TDBCtrlGrid
PanelHeight	published		TDBCtrlGrid
PanelIndex	public		TDBCtrlGrid
PanelWidth	published		TDBCtrlGrid
Parent	public		TControl
ParentColor	published	True	TControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
RowCount	published		TDBCtrlGrid
SelectedColor	published	clWindow	TDBCtrlGrid
ShowFocus	published	True	TDBCtrlGrid
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

AllowDelete

AllowDelete determines whether the user can delete the current record from the dataset by pressing *Ctrl+Delete*.

```
property AllowDelete: Boolean;
```

D

Description Set *AllowDelete* to *False* to prevent the *TDBCtrlGrid* object from deleting records.

When *AllowDelete* is *False*, *TDBCtrlGrid* won't delete records from the dataset when the user presses *Ctrl+Delete* or the *DoKey* method is called with the *Key* parameter set to *gkDelete*. The *TDBCtrlGrid* object can still be used to insert records.

When *AllowDelete* is *True* (the default), *TDBCtrlGrid* will delete the current record when the user presses *Ctrl+Delete* or when the *DoKey* method is called with the Key parameter set to *gkDelete*. Even with *AllowDelete* set to *True*, records can't be deleted unless the dataset is in edit mode.

See also *DoKey* method

AllowInsert

AllowInsert determines whether the user can insert new records with the *Insert* key or append new records with the *Ctrl+Insert* key combination.

```
property AllowInsert: Boolean;
```

Description Set *AllowInsert* to *False* to prevent the *TDBCtrlGrid* object from inserting records.

When *AllowInsert* is *False*, *TDBCtrlGrid* won't insert records into the dataset when the user presses *Insert* or *Ctrl+Insert* or when the *DoKey* method is called with the *Key* parameter set to *gkInsert or gkAppend*. The *TDBCtrlGrid* object can still be used to delete records.

When *AllowInsert* is *True* (the default), *TDBCtrlGrid* will insert a new record into the dataset before the current record when the user presses *Insert* or append a new record to the end of the dataset when the user presses *Ctrl+Insert*. Even with *AllowInsert* set to *True*, records can't be inserted unless the dataset is in edit mode.

See also *DoKey* method

Canvas

Canvas is the drawing surface used to draw the *TDBCtrlGridObject*.

```
property Canvas: TCanvas;
```

Description Use the properties and methods of *Canvas* to draw an image of a panel in the *TDBCtrlGrid* object from an *OnPaintPanel* event handler.

See also *OnPaintPanel* event, *TCanvas* object

ColCount

ColCount specifies the number of columns of panels shown in the grid.

```
property ColCount: Integer;
```

Description Set *ColCount* to the number of columns of panels that should appear in the *TDBCtrlGrid* object. Each panel represents a separate record in the dataset.

Setting *ColCount* causes the *TDBCtrlGrid* object to resize to accommodate the new number of columns. *PanelWidth* doesn't change.

See also *PanelWidth* property, *RowCount* property

DataSource

DataSource identifies the link to the dataset where the *TDBCtrlGrid* object finds its data.

```
property DataSource: TDataSource;
```

Description Set *DataSource* to the *TDataSource* object that links to the dataset where the grid should fetch its data. *DataSource* allows the data-aware grid to read from, write to, and navigate around the dataset, while maintaining record currency with other data-aware objects that use the same data source.

See also *TDataSource* object

EditMode

EditMode determines whether the *TDBCtrlGrid* can be used to insert, delete, or edit records.

```
property EditMode: Boolean;
```

Description Set *EditMode* to *True* to allow the user to edit the dataset using the grid. The user can toggle *EditMode* by pressing *F2* or *Enter*. Inserting or deleting records using the grid will set *EditMode* to *True*.

See also *AllowDelete* property, *AllowInsert* property, *DataSource* property, *DoKey* method

Orientation

Orientation determines the order in which records from the dataset are displayed in the *TDBCtrlGrid*.

```
type TDBCtrlGridOrientation = (goVertical, goHorizontal);

property Orientation: TDBCtrlGridOrientation;
```

Description Set *Orientation* to *goVertical* (the default) to have the panels ordered in rows with a vertical scroll bar. Set *Orientation* to *goHorizontal* to arrange panels in columns with a horizontal scroll bar.

See also *ColCount* property, *RowCount* property

PanelBorder

PanelBorder determines whether each panel is drawn with a raised border around it.

```
type TDBCtrlGridBorder = (gbNone, gbRaised);
property PanelBorder: TDBCtrlGridBorder;
```

Description The default value of *gbRaised* draws a raised border. Set *PanelBorder* to *gbNone* to eliminate the border.

See also *OnPaintPanel* event

PanelCount

PanelCount is the number of records visible in the grid.

```
property PanelCount: Integer;
```

Description *TDBCtrlGrid* can represent up to *ColCount* * *RowCount* panels. Usually, *PanelCount* will be this value. *PanelCount* will be smaller than this if there are not enough records in the dataset to fill the entire grid.

See also *ColCount* property, *RowCount* property

PanelHeight

PanelHeight is the height in pixels of each panel.

```
property PanelHeight: Integer;
```

Description Set *PanelHeight* to the desired size of each panel in the grid. When the *PanelHeight* property is set, the grid resizes itself to fit the new panel height.

See also *PanelWidth* property

PanelIndex

PanelIndex is the index of the panel in the grid which represents the current record.

```
property PanelIndex: Integer;
```

Description Read *PanelIndex* to determine which panel represents the current record in the dataset. Set *PanelIndex* to move in the dataset to the record corresponding to a panel in the grid.

PanelIndex is a number in the range 0..*PanelCount* – 1, where 0 is the first panel, 1 is the second panel, and so on. The location of the panel with the index *PanelIndex* depends on the value of the *Orientation* property. When *Orientation* is *goVertical*, the panels are counted by rows. When *Orientation* is *goHorizontal*, the panels are counted by columns.

See also *Orientation* property, *PanelCount* property

PanelWidth

PanelWidth is the width in pixels of each panel.

property PanelWidth: Integer;

Description Set *PanelWidth* to the desired size of each panel in the grid. When the *PanelWidth* property is set, the grid resizes itself to fit the new panel width.

See also *PanelHeight* property

RowCount

RowCount is the number of rows of panels displayed in the grid.

property RowCount: Integer;

Description Set *RowCount* to the number of rows of panels that should appear in the *TDBCtrlGrid* object. Each panel represents a separate record in the dataset.

Setting *RowCount* causes the *TDBCtrlGrid* object to resize to accommodate the new number of rows. *PanelHeight* doesn't change.

See also *ColCount* property, *PanelHeight* property

ShowFocus

ShowFocus indicates whether *TDBCtrlGrid* should draw a focus rectangle around the current record in the *TDBCtrlGrid* when it has input focus.

property ShowFocus: Boolean;

Description Set *ShowFocus* to *True* to have *TDBCtrlGrid* draw a focus rectangle around the current record when it has input focus. Set *ShowFocus* to *False* when providing visual feedback of focus in some other manner in an *OnPaintPanel* event handler.

See also *OnPaintPanel* event, *PanelIndex* property, *SelectedColor* property

SelectedColor

SelectedColor is the background color of the panel that represents the current record in the dataset.

property SelectedColor: TColor;

Description Set *SelectedColor* to a value other than value of the *Color* property to make the panel that represents the current record stand out. *TDBCtrlGrid* draws the selected panel using *SelectedColor* even when it does not have input focus. All other panels are drawn with the background set to the value of the *Color* property.

SelectedColor can have one of the following values:

Value	Meaning
clBlackclBlack	Black
clMaroon	Maroon
clGreen	Green
clOlive	Olive green
clNavy	Navy blue
clPurple	Purple
clTeal	Teal
clGray	Gray
clSilver	Silver
clRed	Red
clLime	Lime green
clBlue	Blue
clFuchsia	Fuchsia
clAqua	Aqua
clWhite	White
clBackground	Current background color of the Windows desktop
clActiveCaption	Current color of the title bar of the active window
clInactiveCaption	Current color of the title bar of inactive windows
clMenu	Current background color of menus
clWindow	Current background color of windows
clWindowFrame	Current color of window frames
clMenuText	Current color of text on menus
clWindowText	Current color of text in windows
clCaptionText	Current color of the text on the title bar of the active window
clActiveBorder	Current border color of the active window
clInactiveBorder	Current border color of inactive windows
clAppWorkSpace	Current color of the application workspace
clHighlight	Current background color of selected text
clHightlightText	Current color of selected text
clBtnFace	Current color of a button face
clBtnShadow	Current color of a shadow cast by a button
clGrayText	Current color of text that is dimmed
clBtnText	Current color of text on a button
clInactiveCaptionText	Current color of the text on the title bar of an inactive window
clBtnHighlight	Current color of the highlighting on a button
cl3DDkShadow	Windows 95 or NT 4.0 only: Dark shadow for three-dimensional display elements

Value	Meaning
cl3DLight	Windows 95 or NT 4.0 only: Light color for three-dimensional display elements (for edges facing the light source)
clInfoText	Windows 95 or NT 4.0 only: Text color for tool tip controls
clInfoBk	Windows 95 or NT 4.0 only: Background color for tool tip controls

See also *Color* property

Events

The following table lists the events for *TDBCtrlGrid*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnPaintPanel	published	TDBCtrlGrid
OnStartDrag	published	TControl

OnPaintPanel

OnPaintPanel occurs when *TDBCtrlGrid* needs to draw a panel in the grid.

```
type TPaintPanelEvent = procedure(DBCtrlGrid: TDBCtrlGrid; Index: Integer) of object;
property OnPaintPanel: TPaintPanelEvent;
```

Description Write an *OnPaintPanel* event handler to customize the drawing of a panel in the *TDBCtrlGrid* object. *TDBCtrlGrid* paints the background using the *Color* or *SelectedColor* property, including the raised border if it is specified by the *PanelBorder* property. Then the *OnPaintPanel* event occurs to allow an event handler to add to the image of the panel.

D

The *DBCtrlGrid* parameter is the grid that contains the panel. The *Index* parameter is the index of the panel being painted, where 0 is the first panel, 1 is the second panel, and so on. Use the *PanelIndex* property to determine whether the panel being drawn is selected.

Use the *Canvas* property to paint the panel. The point (0,0) on the canvas is the upper left corner of the panel, and the point (*PanelWidth, PanelHeight*) is the lower right corner.

Controls placed on the panel are drawn separately and do not need to be drawn in an *OnPaintpanel* event handler.

See also *Canvas* property, *Color* property, *PanelBorder* property, *PaneHeight* property, *PanelIndex* property, *PanelWidth* property, *SelectedColor* property

Methods

The following table lists the methods for *TDBCtrlGrid*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDBCtrlGrid
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDBCtrlGrid
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DoKey	public	TDBCtrlGrid
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl

Method	Visibility	Documented in
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TDBCtrlGrid
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
KeyDown	public	TDBCtrlGrid
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl

Method	Visibility	Documented in
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TDBCtrlGrid*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to create and initialize a *TDBCtrlGrid* object at runtime. *TDBCtrlGrid* objects placed in a form at design time are created automatically.

Create allocates the helper objects it uses and initializes

- *AllowInsert* and *AllowDelete* to *True*.
- *ColCount* to 1 and *RowCount* to 3.
- *ControlStyle* to [*csOpaque*, *csDoubleClicks*].
- *PanelBorder* to *gbRaised*.
- *PanelWidth* to 200 and *PanelHeight* to 72.
- *SelectedColor* to *Color*.
- *ShowFocus* to *True*.
- *TabStop* to *True*.

See also *AllowInsert* property, *AllowDelete* property, *ColCount* property, *ControlStyle* property, *PanelBorder* property, *PanelHeight* property, *PanelWidth* property, *RowCount* property, *SelectedColor* property, *ShowFocus* property, *TabStop* property

Destroy

Destroy destroys an instance of *TDBCtrlGrid*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TDBCtrlGrid* object is not already destroyed, and only then calls *Destroy*.

Destroy frees the helper objects used by *TDBCtrlGrid* before calling the inherited destructor.

See also *Free* method

DoKey

DoKey implements the responses to a number of standard keystrokes.

```
type TDBCtrlGridKey = (gkNull, gkEditMode, gkPriorTab, gkNextTab, gkLeft, gkRight, gkUp,
  gkDown, gkScrollUp, gkScrollDown, gkPageUp, gkPageDown, gkHome, gkEnd, gkInsert,
  gkAppend, gkDelete, gkCancel);

procedure DoKey(Key: TDBCtrlGridKey);
```

Description Call *DoKey* to perform the operation specified by the *Key* parameter. For example, use *DoKey* to remap keystrokes from an *OnKeyDown* event handler.

The actions performed for the different values of *Key* are given in the following table:

Value	Action
gkNull	Do nothing.
gkEditMode	Toggle the *EditMode* property.
gkPriorTab	Move to the previous panel.
gkNextTab	Move to the next panel.
gkLeft	Move one panel to the left.
gkRight	Move one panel to the right.
gkUp	Move one panel up.
gkDown	Move one panel down.
gkScrollUp	Move one panel up.
gkScrollDown	Move one panel down.
gkPageUp	Move *ColCount* * *RowCount* records up in the dataset.
gkPageDown	Move *ColCount* * *RowCount* records down in the dataset.
gkHome	Move to the first record in the dataset.
gkEnd	Move to the last record in the dataset.
gkInsert	Insert a new record prior to the current record in the dataset and set *EditMode* to *True*.
gkAppend	Insert a new record at the end of the dataset and set *EditMode* to *True*.
gkDelete	Delete the current record from the dataset and set *EditMode* to *False*.
gkCancel	Cancel any edits not written to the dataset and set *EditMode* to *False*.

See also *AllowDelete* property, *AllowInsert* property, *EditMode* property, *KeyDown* method, *OnKeyDown* event

GetTabOrderList

GetTabOrderList overrides the default to remove all child controls from the tab order.

```
procedure GetTabOrderList(List: TList); override;
```

D

Description *GetTabOrderList* returns an empty list. *TDBCtrlGrid* implements its own handling of the tab key to move through the panels in database order.

See also *TabOrder* property

KeyDown

KeyDown performs special processing when a key is pressed.

```
procedure KeyDown(var Key: Word; Shift: TShiftState); override;
```

Description After calling the *OnKeyDown* event handler, *KeyDown* translates each combination of the *Key* and *Shift* parameters into a logical key code and calls the *DoKey* method to perform the appropriate action.

KeyDown performs the following translations:

Key and Shift values	Key code
LeftArrow	gkLeft
RightArrow	gkRight
UpArrow	gkUp
DownArrow	gkDown
PageUp	gkPageUp
PageDown	gkPageDown
Home	gkHome
End	gkEnd
Return	gkEditMode
F2	gkEditMode
Insert	gkAppend
Ctrl+Insert	gkInsert
Ctrl+Delete	gkDelete
Escape	gkCancel
All other keys	gkNull

See also *DoKey* method, *OnKeyDown* event

TDBCtrlPanel

Unit: DBCGrids

TObject • TPersistent • TComponent • TControl • TWinControl • TDBCtrlPanel

TDBCtrlPanel is a panel in a *TDBCtrlGrid* object.

Description Use *TDBCtrlPanel* to hold the fields from a single record in a dataset. *TDBCtrlGrid* represents the records in a dataset by arranging copies of the

same *TDBCtrlPanel* object in rows or columns, where each panel represents a single record in the dataset.

Lay out a single panel at design time. Add data-aware edit controls or data display objects. At runtime, *TDBCtrlGrid* replicates the panel for the records in the dataset.

The appearance of the panel is determined by the properties of the *TDBCtrlGrid* object that contains the panel. The *OnPaintPanel* event of *TDBCtrlGrid* allows customized painting of the panels.

See also *TDBCtrlGrid* object, *TDBEdit* object, *TDBImage* object, *TDBMemo* object, *TDBRichEdit* object, *TDBText* object

Properties

The following table lists the properties of *TDBCtrlPanel*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Cursor	published	crDefault	TControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent

D

Property	Visibility	Default	Documented in
Parent	public		TControl
ParentWindow	public		TWinControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	public	False	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

Methods

The following table lists the methods of *TDBCtrlPanel*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TWinControl
CreateLinked	public	TDBCtrlPanel
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TWinControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject

Method	Visibility	Documented in
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl

Method	Visibility	Documented in
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

CreateLinked

CreateLinked creates and initializes an instance of *TDBCtrlPanel*.

constructor CreateLinked(DBCtrlGrid: TDBCtrlGrid);

Description Do not call the constructor for *TDBCtrlPanel* directly. *TDBCtrlGrid* calls the *CreateLinked* method for *TDBCtrlPanel* from its own constructor.

CreateLinked initializes the *Parent* property to the *DBCtrlGrid* parameter and initializes the *ControlStyle* property to [*csAcceptsControls, csCaptureMouse, csClickEvents, csDoubleClicks, csOpaque, csReplicatable*].

See also *ControlStyle* property, *Parent* property, *Create* method

TDBDataSet

Unit: DBTables

TObject • TPersistent • TComponent • TDataSet • TBDEDataSet • TDBDataSet

Encapsulates database connectivity for descendent dataset objects.

Description *TDBDataSet* is a dataset object that defines database-related connectivity properties and methods for a dataset. Applications never use *TDBDataSet* objects directly. Instead they use the descendants of *TDBDataSet*, such as *TQuery*, *TStoredProc*, and *TTable*, which inherit its database-related properties and methods.

Developers who create custom dataset components may want to derive them from *TDBDataSet* to inherit all the functionality of *TDataSet* and the BDE-related properties and methods of *TBDEDataSet* in addition to the database-specific properties of *TDBDataSet*. In this case, developers should examine the source code to study the protected methods of *TDBDataSet* that are not documented for this object.

See also *TDataSet* object, *TBDEDataSet* object, *TQuery* object, *TStoredProc* object, *TTable* object

Properties

The following table lists the properties of *TDBDataSet*:

Property	Visibility	Default	Documented in
Active	published	False	TDataSet
AutoCalcFields	published	True	TDataSet
BOF	public, read-only		TDataSet
Bookmark	public		TDataSet
CacheBlobs	public	True	TBDEDataSet
CachedUpdates	published	False	TBDEDataSet
CanModify	public, read-only		TBDEDataSet
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
Database	public, read-only		TDBDataSet
DatabaseName	published		TDBDataSet
DataSource	public, read-only		TDataSet
DBHandle	public, read-only		TDBDataSet
DBLocale	public, read-only		TDBDataSet
DBSession	public, read-only		TDBDataSet
DefaultFields	public, read-only		TDataSet
Designer	public, read-only		TDataSet
DesignInfo	public		TComponent
EOF	public, read-only		TDataSet
ExpIndex	public, read-only		TBDEDataSet
FieldCount	public, read-only		TDataSet
FieldDefs	public		TDataSet
Fields	public		TDataSet
FieldValues	public		TDataSet
Filter	published		TBDEDataSet
Filtered	published	False	TBDEDataSet
FilterOptions	published	[]	TBDEDataSet
Found	public, read-only		TDataSet
Handle	public, read-only		TBDEDataSet
KeySize	public, read-only		TBDEDataSet
Locale	public, read-only		TBDEDataSet
Modified	public, read-only		TDataSet
Name	published		TDataSet
Owner	public, read-only		TComponent

Property	Visibility	Default	Documented in
Provider	public, read-only		TDBDataSet
RecNo	public		TBDEDataSet
RecordCount	public, read-only		TBDEDataSet
RecordSize	public, read-only		TBDEDataSet
SessionName	published		TDBDataSet
State	public, read-only		TDataSet
Tag	published	0	TComponent
UpdateObject	public		TBDEDataSet
UpdateRecordTypes	public		TBDEDataSet
UpdatesPending	public, read-only		TBDEDataSet
VCLComObject	public		TComponent

Database

Identifies the database component for which this dataset represents one or more tables.

```
property Database: TDatabase;
```

Description Use *Database* to access the properties, events, and methods of the database component associated with this dataset. *Database* is a read-only property that is automatically set when the database specified by the *DatabaseName* property is opened.

See also *DatabaseName* property

DatabaseName

Specifies the name of the database to associate with this dataset.

```
property DatabaseName: string;
```

Description Use *DatabaseName* to specify the name of the database to associate with this dataset component. *DatabaseName* should match the name of a database component used in the application.

Note Attempting to set *DatabaseName* when a database already associated with this component is open raises an exception.

Tip At design time double-click a *TDatabase* component to invoke the Database editor and set the *DatabaseName*.

See also *Database* property, *DBHandle property*

DBHandle

Specifies the Borland Database Engine (BDE) database handle for the dataset.

```
type HDBISES: Longint;
```

```
property DBHandle: HDBISES;
```

Description Check *DBHandle* to determine the database handle for the dataset. *DBHandle* is only useful in applications that bypass database and dataset methods to write directly to the BDE API. Many BDE function calls require a handle parameter. *DBHandle* is assigned an initial value when a session is activated.

Note Do not use this property unless an application requires BDE functionality not available through standard Delphi components.

See also *DBLocale* property, *TDatabase* object

DBLocale

Identifies the Borland Database Engine (BDE) language driver for the dataset component.

```
type
  TLocale: Pointer;

property DBLocale: TLocale;
```

Description Examine *DBLocale* to determine the BDE language driver used by the database component associated with which this dataset is associated. Applications that make direct calls to the BDE may need to pass *DBLocale* information as an API function parameter.

Note Do not use this property unless an application requires BDE functionality not available through standard Delphi components.

See also *DBHandle* property, *TDatabase* object

DBSession

Points to the database session component with which this dataset component is associated.

```
property DBSession: TSession
```

Description Use *DBSession* to determine the session component that controls the database component with which this dataset component is associated. By default, a database component is associated with the default session component, *Session*, that is automatically created for all database applications.

See also *SessionName* property, *TDatabase* object, *TSession* object

Provider

Identifies the provider interface for this dataset.

```
property Provider: IProvider;
```

Description Check *Provider* to determine the *IProvider* interface for this object. The *IProvider* interface is only of interest to developers of multi-tiered

applications who wish to enable client datasets to communicate directly with this dataset object on the application server through the *IProvider* interface.

See also *Data* property

SessionName

Identifies the name of the session with which this dataset is associated.

property SessionName: **string**;

Description Use *SessionName* to determine or specify the session with which a dataset component is associated. *SessionName* is automatically set to the name of the *SessionName* property of the database component with which a dataset component is associated. If *SessionName* is blank, a dataset component is automatically associated with the default session, *Session*.

To associate a dataset component with a different session *SessionName* must matches the *SessionName* property of an existing session component that is also used by the database component with which this dataset is associated.

See also *DBSession* property

Events

The following table lists the events of *TDBDataSet*:

Event	Visibility	Documented in
AfterCancel	published	TDataSet
AfterClose	published	TDataSet
AfterDelete	published	TDataSet
AfterEdit	published	TDataSet
AfterInsert	published	TDataSet
AfterOpen	published	TDataSet
AfterPost	published	TDataSet
AfterScroll	published	TDataSet
BeforeCancel	published	TDataSet
BeforeClose	published	TDataSet
BeforeDelete	published	TDataSet
BeforeEdit	published	TDataSet
BeforeInsert	published	TDataSet
BeforeOpen	published	TDataSet
BeforePost	published	TDataSet
BeforeScroll	published	TDataSet
OnCalcFields	published	TDataSet
OnDeleteError	published	TDataSet
OnEditError	published	TDataSet

Event	Visibility	Documented in
OnFilterRecord	published	TDataSet
OnNewRecord	published	TDataSet
OnPostError	published	TDataSet
OnServerYield	published	TBDEDataSet
OnUpdateError	published	TBDEDataSet
OnUpdateRecord	published	TBDEDataSet

Methods

The following table lists the methods of *TDBDataSet*:

Method	Visibility	Documented in
ActiveBuffer	public	TDataSet
Append	public	TDataSet
AppendRecord	public	TDataSet
ApplyUpdates	public	TBDEDataSet
Assign	public	TPersistent
BookmarkValid	public	TBDEDataSet
Cancel	public	TBDEDataSet
CancelUpdates	public	TBDEDataSet
CheckBrowseMode	public	TDataSet
CheckOpen	public	TDBDataSet
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClearFields	public	TDataSet
Close	public	TDataSet
CommitUpdates	public	TBDEDataSet
CompareBookmarks	public	TBDEDataSet
ControlsDisabled	public	TDataSet
Create	public	TBDEDataSet
CursorPosChanged	public	TDataSet
DefaultHandler	public	TObject
Delete	public	TDataSet
Destroy	public	TBDEDataSet
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableControls	public	TDataSet

Method	Visibility	Documented in
Dispatch	public	TObject
Edit	public	TDataSet
EnableControls	public	TDataSet
FetchAll	public	TBDEDataSet
FieldAddress	public	TObject
FieldByName	public	TDataSet
FindComponent	public	TComponent
FindField	public	TDataSet
FindFirst	public	TDataSet
FindLast	public	TDataSet
FindNext	public	TDataSet
FindPrior	public	TDataSet
First	public	TDataSet
Free	public	TObject
FreeBookmark	public	TDataSet
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetBookmark	public	TDataSet
GetCurrentRecord	public	TBDEDataSet
GetFieldList	public	TDataSet
GetFieldNames	public	TDataSet
GetIndexInfo	public	TBDEDataSet
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GotoBookmark	public	TDataSet
HasParent	public	TComponent
InheritsFrom	public	TObject
InitInstance	public	TObject
Insert	public	TDataSet
InsertComponent	public	TComponent
InsertRecord	public	TDataSet
InstanceSize	public	TObject
IsEmpty	public	TDataSet
IsLinkedTo	public	TDataSet
Last	public	TDataSet
Locate	public	TBDEDataSet
Lookup	public	TBDEDataSet
MethodAddress	public	TObject

Method	Visibility	Documented in
MethodName	public	TObject
MoveBy	public	TDataSet
NewInstance	public	TObject
Next	public	TDataSet
Open	public	TDataSet
Post	public	TDataSet
Prior	public	TDataSet
Refresh	public	TDataSet
RemoveComponent	public	TComponent
Resync	public	TDataSet
RevertRecord	public	TBDEDataSet
SafeCallException	public	TComponent
SetFields	public	TDataSet
Translate	public	TDataSet
UpdateCursorPos	public	TDataSet
UpdateRecord	public	TDataSet
UpdateStatus	public	TBDEDataSet

CheckOpen

Checks the result of a call to the Borland Database Engine (BDE).

```
function CheckOpen(Status: DBIResult): Boolean;
```

Description Call *CheckOpen* to determine if a call to the BDE returns an error when an attempt is made to access a table. *Status* is the return result of a previous call to the BDE. *CheckOpen* returns *True* if there is access is successful. If *Status* indicates insufficient table rights when accessing a Paradox table, *CheckOpen* calls the database session's *GetPassword* method to prompt the user for a password. If the dialog is successful, *CheckOpen* returns *True*.

Otherwise *CheckOpen* returns *False*, indicating that dataset access failed.

See also *TDatabase* object, *TSession* object, *TSession.GetPassword* method

TDBEdit

Unit: DBCtrls — Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomEdit • TCustomMaskEdit • TDBEdit

TDBEdit is a single-line edit control that can display and edit a field in a dataset.

D

Description Use *TDBEdit* to enable users to edit a database field. *TDBEdit* can use a mask to restrict input to data that is valid for the database field. *TDBEdit* uses the *Text* property to represent the contents of the field.

TDBEdit permits only a single line of text. If the field may contain lengthy data that would require multiple lines, consider using a *TDBMemo* object.

If the application doesn't require the data-aware capabilities of *TDBEdit*, use an edit control (*TEdit*) or a masked edit control (*TMaskEdit*) instead, to conserve system resources.

See also T*DataSet* object, T*DataSource* object, *TDBMemo* object, *TEdit* object, *TField* object, *TMaskEdit* object

Properties

The following table lists the properties of *TDBEdit*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
AutoSelect	published	True	TCustomEdit
AutoSize	published	True	TCustomEdit
BorderStyle	published	bsSingle	TCustomEdit
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
CharCase	published	ecNormal	TCustomEdit
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DataField	published		TDBEdit
DataSource	published		TDBEdit

Property	Visibility	Default	Documented in
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
EditText	public		TCustomMaskEdit
Enabled	published	True	TControl
Field	public, read-only		TDBEdit
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
IsMasked	public, read-only		TCustomMaskEdit
Left	published		TControl
MaxLength	published	0	TCustomMaskEdit
Modified	public		TCustomEdit
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PasswordChar	published	#0	TCustomEdit
PopupMenu	published		TControl
ReadOnly	published	False	TDBEdit
SelLength	public		TCustomEdit
SelStart	public		TCustomEdit
SelText	public		TCustomEdit
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	public		TCustomMaskEdit
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

D

DataField

DataField specifies the field from which the edit control displays data.

```
property DataField: string;
```

Description Use *DataField* to bind the edit control to a field in the dataset. To fully specify a database field, both the dataset and the field within that dataset must be defined. The *DataSource* property of the edit box control specifies the dataset which contains the *DataField*.

If the *DataField* refers to a database field that contains integer or floating-point data, only characters that are valid in such a field can be entered in the edit box. Characters that are not legal are not accepted.

See also *DataSource* property, *Field* property

DataSource

DataSource links the edit control to the dataset that contains the field it represents.

```
property DataSource: TDataSource;
```

Description Use *DataSource* to link the edit control to a dataset in which the data can be found. To fully specify a database field for the edit control, both the dataset and a field within that dataset must be defined. Use the *DataField* property to specify the particular field within the dataset.

See also *DataField* property, *Field* property

Field

Field specifies the *TField* object for the database field the edit box represents.

```
property Field: TField;
```

Description Read *Field* to get direct access to the contents and properties of the database field without going through the edit window. Use *Field* to change the contents of the database field programmatically. Use the *Text* property instead if you want to change the contents of the edit window without immediately posting the changes to the dataset. Changing the *Text* property allows the user to cancel the changes.

See also *DataField* property, *DataSource* property, *Reset* method, *Text* property, *TField* object

ReadOnly

ReadOnly determines whether the user can use the database edit box to change the value of the field in the current record.

```
property ReadOnly: Boolean;
```

Description Set *ReadOnly* to *True* to prevent the user from changing the contents of the database field. When *ReadOnly* is *True*, the edit control is used only to display the contents of the field. If *ReadOnly* is *False*, the user can change the field's value as long as the dataset is in edit mode.

See also *Change* method, *EditCanModify* method, *Field* property

Events

The following table lists the events of *TDBEdit*:

Event	Visibility	Documented in
OnChange	published	TCustomEdit
OnClick	published	TControl
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TDBEdit*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
Change	protected	TDBEdit
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject

Method	Visibility	Documented in
CleanupInstance	public	TObject
Clear	public	TCustomMaskEdit
ClearSelection	public	TCustomEdit
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
CopyToClipboard	public	TCustomEdit
Create	public	TDBEdit
CreateParented	public	TWinControl
CutToClipboard	public	TCustomEdit
DefaultHandler	public	TObject
Destroy	public	TDBEdit
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EditCanModify	protected	TDBEdit
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetSelTextBuf	public	TCustomEdit
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TCustomMaskEdit
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject

Method	Visibility	Documented in
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
KeyDown	protected	TDBEdit
KeyPress	protected	TDBEdit
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TDBEdit
PaintTo	public	TWinControl
PasteFromClipboard	public	TCustomEdit
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
Reset	protected	TDBEdit
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SelectAll	public	TCustomEdit
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetSelTextBuf	public	TCustomEdit
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl
ValidateEdit	public	TCustomMaskEdit

Change

Change keeps track of the fact that the text has been modified, before letting the inherited method call the *OnChange* event handler if it is assigned.

```
procedure Change; override;
```

D

Description *Change* is a protected method. Applications cannot call it directly. It is triggered automatically by Windows' events. *Change* responds to changes in the text by internally keeping track of the fact that the text has been modified. This information can then be used to know whether there are any changes to post to the dataset or to remove if the user requests a reset.

See also *OnChange* event, *Reset* method, *Text* property

Create

Create creates an instance of *TDBEdit*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call create to create a *TDBEdit* object at runtime. *TDBEdit* objects placed on a form at design time are created automatically.

Create overrides the inherited constructor to initialize the database handling capabilities of the edit control.

See also *TCustomMaskEdit.Create* method

Destroy

Destroy destroys an instance of *TDBEdit*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TDBEdit* object is not already freed and only then calls *Destroy*.

Destroy overrides the inherited destructor to free the helper objects that were allocated to manage display issues and database interactions.

See also *Free* method

EditCanModify

EditCanModify allows or disallows editing of the text depending on whether the dataset is in edit mode.

```
function EditCanModify: Boolean; override;
```

Description *EditCanModify* overrides the inherited method so that the control will only allow edits when the dataset is in edit mode. If the edit control should never allow edits, the *ReadOnly* property should be set.

Note *EditCanModify* is a protected method. Applications cannot call it directly.

See also *DataSource* property, *ReadOnly* property, *Text* property

KeyDown

KeyDown supplements the inherited method to drive the dataset into edit mode when the user presses *Del* or *Shift+Insert*.

```
type TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

procedure KeyDown(var Key: Word; Shift: TShiftState); override;
```

Description KeyDown first calls the inherited method, which calls the *OnKeyDown* event handler. *KeyDown* next attempts to put the dataset into an editing state when the user types *Del* or *Shift+Insert*. Derived classes can override *KeyDown* to handle additional key combinations, but the inherited method should always be called.

Note *KeyDown* is a protected method. Applications cannot call it directly. It is called automatically in response to Windows events.

See also *DataSource* property, *EditCanModify* method, *KeyPress* method, *OnKeyDown* event, *TCustomMaskEdit.KeyDown* method

KeyPress

KeyPress supplements the inherited method to handle any keystrokes that alter the state of the edit control.

```
procedure KeyPress(var Key: Char); override;
```

Description *KeyPress* first calls the inherited method, which calls the *OnKeyPress* event handler. Next, *KeyPress* discards the keystroke if it is not a valid character for the field type, or if there is no link to a database field. Valid characters cause *KeyPress* to attempt to put the dataset into edit mode.

Pressing the *Escape* key causes *KeyPress* to call *Reset*, throwing away all unposted edits.

Note *KeyPress* is a protected method. Applications cannot call it directly. It is called automatically in response to Windows events.

See also *DataSource* property, *EditCanModifyMethod*, *KeyDown* method, *OnKeyPress* event, *Reset* method, *TCustomMaskEdit.KeyPress* method

Notification

Notification overrides the inherited method to respond to *opRemove* notifications on the *DataSource*.

```
type TOperation = (opInsert, opRemove);

procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description After calling the inherited method, *Notification* checks whether the notification indicates the removal of the *DataSource*. If so, *Notification* sets the *DataSource* property to **nil**.

Derived classes can override *Notification* method to handle additional notifications, but the inherited method should be always be called.

Note *Notification* is a protected method. Applications cannot call it directly.

See also *DataSource* property

Reset

Reset cancels all unposted edits.

procedure Reset; **override**;

Description *Reset* overrides the default method to cancel all unposted edits to the dataset when the user presses the *Escape* key.

Note *Reset* is a protected method. Applications cannot call it directly. It is called automatically when the user presses the *Escape* key.

See also *Change* method, *KeyPress* method

TDBGrid

Unit: DBGrids | Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TCustomGrid • TCustomDBGrid • TDBGrid

TDBGrid displays and manipulates records from a dataset in a tabular grid.

Description Put a *TDBGrid* object on a form to display and edit the records from a database table or query. Applications can use the data grid to insert, delete, or edit data in the database, or simply to display it.

At runtime, users can use the database navigator (*TDBNavigator*) to move through data in the grid, and to insert, delete, and edit the data. Edits that are made in the data grid are not posted to the underlying dataset until the user moves to a different record or closes the application.

TDBGrid implements the generic behavior introduced in *TCustomDBGrid*. *TDBGrid* publishes many of the properties inherited from *TCustomDBGrid*, but does not introduce any new behavior.

See also *TDBCtrlGrid* object, *TDBNavigator* object

Properties

The following table lists the properties of *TDBGrid*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
BorderStyle	published	bsSingle	TCustomGrid

Property	Visibility	Default	Documented in
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
Columns	published		TCustomDBGrid
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DataSource	published		TCustomDBGrid
DefaultDrawing	published	True	TCustomDBGrid
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
EditorMode	public		TCustomGrid
Enabled	published	True	TControl
FieldCount	public, read-only		TCustomDBGrid
Fields	public, read-only		TCustomDBGrid
FixedColor	published	clBtnFace	TCustomGrid
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
Left	published		TControl
Name	published		TControl

Property	Visibility	Default	Documented in
Options	published	[dgEditing, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgConfirmDelete, dgCancelOnExit]	TCustomDBGrid
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ReadOnly	published	False	TCustomDBGrid
SelectedField	public		TCustomDBGrid
SelectedIndex	public		TCustomDBGrid
SelectedRows	public, read-only		TCustomDBGrid
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
TitleFont	published		TCustomDBGrid
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

Events

The following table lists the events of *TDBGrid:*

Event	Visibility	Documented in
OnCellClick	published	TCustomDBGrid
OnColEnter	published	TCustomDBGrid
OnColExit	published	TCustomDBGrid
OnColumnMoved	published	TCustomDBGrid
OnDblClick	published	TControl

Event	Visibility	Documented in
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnDrawColumnCell	published	TCustomDBGrid
OnDrawDataCell	published	TCustomDBGrid
OnEditButtonClick	published	TCustomDBGrid
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnStartDrag	published	TControl
OnTitleClick	published	TCustomDBGrid

Methods

The following table lists the methods of *TDBGrid*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TCustomDBGrid
CreateParented	public	TWinControl
DefaultDrawColumnCell	public	TCustomDBGrid
DefaultDrawDataCell	public	TCustomDBGrid
DefaultHandler	public	TObject
Destroy	public	TCustomDBGrid
DestroyComponents	public	TComponent
Destroying	public	TComponent

Method	Visibility	Documented in
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
MouseCoord	public	TCustomGrid
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl

Method	Visibility	Documented in
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl
ValidFieldIndex	public	TCustomDBGrid

TDBImage

Unit: DBCtrls | Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TDBImage

The *TDBImage* control displays a graphic image from a BLOB (binary large object) field of the current record of a dataset.

Description Use *TDBImage* to represent the value of graphic fields. *TDBImage* allows a form to display graphical data from a dataset. *TDBImage* can allow the user to edit graphical data using the clipboard. While the application is running and the database image control has focus, users can cut, copy, and paste using the standard windows accelerator keys (*Ctrl+X*, *Ctrl+C*, and *Ctrl+V*). Users can return to the original state of the database image control by pressing *Esc* before moving to another record.

If the application doesn't require the data-aware capabilities of *TDBImage*, use an image control (*TImage*) instead to conserve system resources.

See also *TDBEdit* object, *TDBMemo* object, *TDBRichEdit* object, *TImage* object

Properties

The following table lists the properties of *TDBImage*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
AutoDisplay	published	True	TDBImage
BorderStyle	published	bsSingle	TDBImage

D

Property	Visibility	Default	Documented in
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Center	published	True	TDBImage
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DataField	published		TDBImage
DataSource	published		TDBImage
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Field	public, read-only		TDBImage
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
Left	published		TControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
Picture	public		TDBImage

Property	Visibility	Default	Documented in
PopupMenu	published		TControl
QuickDraw	published	True	TDBImage
ReadOnly	published	False	TDBImage
ShowHint	published		TControl
Showing	public, read-only		TWinControl
Stretch	published	False	TDBImage
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

AutoDisplay

AutoDisplay determines whether to automatically display the contents of a graphic BLOB in the database image control.

property AutoDisplay: Boolean;

Description If *AutoDisplay* is *True* (the default value), the image automatically displays new data when the underlying BLOB field changes (such as when moving to a new record).

If *AutoDisplay* is *False*, the image clears whenever the underlying BLOB field changes. To display the data, the user can double-click on the control or select it and press *Enter*. In addition, calling the *LoadPicture* method ensures that the control is showing data.

Change the value of *AutoDisplay* to *False* if the automatic loading of BLOB fields seems to take too long.

See also *LoadPicture* method

BorderStyle

BorderStyle determines whether the database image control has a border.

property BorderStyle: TBorderStyle;

D

Description Set BorderStyle to specify whether a single line border should be drawn around the database image control. These are the possible values:

Value	Meaning
bsNone	No visible border
bsSingle	Single-line border

See also *Ctl3D* property

Center

Center determines whether the image is centered in the image control.

```
property Center: Boolean;
```

Description When the image does not fit perfectly within the image control, use *Center* to specify how the image is positioned. When *Center* is *True*, the image is centered in the control. When *Center* is *False*, the upper left corner of the image is positioned at the upper left corner of the control.

Note *Center* has no effect if the *Stretch* property is *True*.

See also *Stretch* property

DataField

DataField specifies the field from which the database image displays data.

```
property DataField: string;
```

Description Use *DataField* to bind the image control to a field in the dataset. To fully specify a database field, both the dataset and the field within that dataset must be defined. The *DataSource* property of the image control specifies the dataset which contains the *DataField*. *DataField* should specify a graphic field.

See also *DataSource* property, *Field* property

DataSource

DataSource links the image control to a dataset.

```
property DataSource: TDataSource;
```

Description Use *DataSource* to link the image control to a dataset in which the data can be found. To fully specify a database field for the image control, both the dataset and a field within that dataset must be defined. Use the *DataField* property to specify the particular field within the dataset.

See also *DataField* property, *TDataSource* object

Field

Field is the *TField* component the database image is linked to.

property Field: TField;

Description Read *Field* to get direct access to the contents and properties of the database field without going through the image control's window. Use *Field* to change the contents of the database field programmatically. Use the *Picture* property to set the value of the field and display it in the image control's window.

See also *DataField* property, *Picture* property, *TField* object

Picture

Picture determines the image that appears on the database image control.

property Picture: TPicture;

Description Read *Picture* to obtain the *TPicture* object that represents the value of the field. Set *Picture* to programmatically change the value of the graphic field. The property value is a *TPicture* object which can contain an icon, metafile, bitmap, or other graphic object.

See also *CutToClipboard* method, *CopyToClipboard* method, *LoadPicture* method, *PasteFromClipboard* method

QuickDraw

QuickDraw specifies whether the image is displayed using a palette.

property QuickDraw: Boolean;

Description Set *QuickDraw* to specify whether a customized palette should be used when displaying field values. If *False*, a palette is used, to provide the best possible image quality at the expense of additional processing time. If *True*, no special palette is used, which is faster, but results in poorer picture quality, especially with 256-color images on a 256-color video driver.

See also *LoadPicture* method

ReadOnly

ReadOnly determines whether the user can change the contents of the field using the image control.

property ReadOnly: Boolean;

Description Set *ReadOnly* to *True* to prevent the user from changing the contents of the database field. When *ReadOnly* is *True*, the image control is used only to display the contents of the field. If *ReadOnly* is *False*, the user can change the field's value as long as the dataset is in edit mode.

See also *DataField* property, *DataSource* property

Stretch

Stretch determines whether bitmaps and metafiles assume the size and shape of the database image.

```
property Stretch: Boolean;
```

Description Set *Stretch* to *True* to cause the picture to assume the size and shape of the image control. When the image control resizes, the picture resizes also. *Stretch* resizes the height and width of the image independently. Thus, unlike a simple change in magnification, stretch can distort the image if the image control is not the same shape as the image.

See also *Picture* property

Events

The following table lists the events of *TDBImage*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TDBImage*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl

Method	Visibility	Documented in
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
CopyToClipboard	public	TDBImage
Create	public	TDBImage
CreateParented	public	TWinControl
CutToClipboard	public	TDBImage
DefaultHandler	public	TObject
Destroy	public	TDBImage
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl

Method	Visibility	Documented in
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
LoadPicture	public	TDBImage
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
PasteFromClipboard	public	TDBImage
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

CopyToClipboard

CopyToClipboard copies the picture in the image control to the Clipboard.

procedure CopyToClipboard;

Description Use *CopyToClipboard* to put a copy of the graphic specified by the *Picture* property in the Clipboard.

See also *CutToClipboard* method, *PasteFromClipboard* method, *Picture* property

Create

Create creates an instance of *TDBImage*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to instantiate a *TDBImage* object at runtime. For *TDBImage* objects placed on forms or data modules at design time, *Create* is called automatically.

After calling the inherited constructor to allocate and initialize a generic control object, *Create*

- Adds *csOpaque, csReplicatable* to the *ControlStyle,*
- Sets *TabStop, AutoDisplay* and *Center* to *True* and sets *BorderStyle* to *bsSingle,*
- Creates a *TPicture* object and assigns it to the *Picture* property,
- Initializes the height and width of the control,
- Creates and initializes the helper objects needed to link to a dataset.

See also *AutoDisplay* property, *Center* property, *ControlStyle* property, *Picture* property, *TabStop* property, *TCustomControl.Create* method, *TFieldDataLink* object

CutToClipboard

CutToClipboard deletes the image in the control and copies it to the Clipboard.

```
procedure CutToClipboard;
```

Description Use the *CutToClipboard* method to delete the image in the control when copying it to the Clipboard. *CutToClipboard* gives the associated field a null value.

See also *CopyToClipboard* method, *PasteFromClipboard* method

Destroy

Destroy destroys an instance of *TDBImage*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TDBImage* object is not already freed and only then calls *Destroy*.

Destroy frees the helper objects of the *TDBImage* before calling the inherited destructor.

See also *Create* method, *Free* method

LoadPicture

LoadPicture loads the image stored in the field into the database image control.

```
procedure LoadPicture;
```

Description If the value of the *AutoDisplay* property is *False*, the image of a database image control is not automatically loaded. If *AutoDisplay* is *False*, control when the image is loaded at runtime by calling *LoadPicture* when the image should appear in the control.

See also *Picture* property, *PasteFromClipboard* method

PasteFromClipboard

PasteFromClipboard copies the contents of the Clipboard to the control.

```
procedure PasteFromClipboard;
```

Description Use *PasteFromClipboard* to set the field value to an image stored in the Clipboard.

See also *CutToClipboard* method, *CopyToClipboard* method

TDBListBox

Unit: DBCtrls | Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomListBox • TDBListBox

The *TDBListBox* control is a data-aware list box that allows the user to change the value of the field of the current record in a dataset by selecting an item from a list.

Description Use *TDBListBox* to add a list box to a form that permits users to change a field value from a fixed set of choices.

If the application doesn't require the data-aware capabilities of *TDBListBox*, use a list box (*TListBox*) instead to conserve system resources.

Component writers wanting to create a customized list box objects should use *TCustomListBox* as a base class.

See also *TDBComboBox* object, *TDBLookupListBox* object, *TListBox* object

Properties

The following table lists the properties of *TDBListBox*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
BorderStyle	published	bsSingle	TCustomListBox
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
Canvas	public, read-only		TCustomListBox
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published	True	TWinControl
Cursor	published	crDefault	TControl
DataField	published		TDBListBox
DataSource	published		TDBListBox
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
Enabled	published	True	TControl
Field	public, read-only		TDBListBox
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
IntegralHeight	published	False	TCustomListBox
ItemHeight	published		TCustomListBox
ItemIndex	public		TCustomListBox
Items	published		TDBListBox
Left	published		TControl
Name	published		TControl

Property	Visibility	Default	Documented in
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ReadOnly	published	False	TDBListBox
SelCount	public, read-only		TCustomListBox
Selected	public		TCustomListBox
ShowHint	published		TControl
Showing	public, read-only		TWinControl
Style	published	lbStandard	TCustomListBox
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
TopIndex	public		TCustomListBox
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

DataField

DataField identifies the field whose value is displayed by the listbox.

property DataField: **string**;

Description Set *DataField* to specify which field is represented by the listbox. The listbox displays the current value of this field, and allows the user to set the value of this field on the current record. The dataset the field is located in is specified by the *DataSource* property.

See also *DataSource* property, *Field* property

DataSource

DataSource links the listbox to a dataset.

property DataSource: TDataSource;

Description Specify the data source component that identifies the dataset the data is found in. To allow the listbox to represent the data for a field, both the *DataSource* and the *DataField* properties must be set.

See also *DataField* property, *Field* property

Field

Field returns the *TField* object the listbox is linked to.

property Field: TField;

Description Use the *Field* object when you want to change the value of the data in the field programmatically.

See also *DataField* property, *DataSource* property, *TField* object

Items

Items contains the strings that appear in the list box.

property Items;

Description Use *Items* to specify the values the user can choose from. Because *Items* is an object of type *TStrings*, you can add, delete, insert, and move items using the *Add*, *Delete*, *Insert*, *Exchange*, and *Move* methods of the *TStrings* object. For example, to add each string to the listbox, write a line of code such as the following:

```
DBListBox1.Items.Add('AFieldValue');
```

The *ItemIndex* property determines which item is selected, if any.

See also *ItemIndex* property

ReadOnly

ReadOnly determines if the user can change the field value using the listbox.

property ReadOnly: Boolean;

Description Set *ReadOnly* to specify whether the listbox should be used for display purposes only. If *ReadOnly* is *True*, the listbox can only be used to display the value of the field on the current record. If *ReadOnly* is *False*, the user can use the listbox to change the field's value to one of the values in the *Items* list.

See also *Items* property, *TDataSet.Edit* method

Events

The following table lists the events of *TDBListBox*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDblClick	published	TControl

D

Event	Visibility	Documented in
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnDrawItem	published	TCustomListBox
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMeasureItem	published	TCustomListBox
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TDBListBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
Clear	public	TCustomListBox
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDBListBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDBListBox
DestroyComponents	public	TComponent
Destroying	public	TComponent

Method	Visibility	Documented in
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
ItemAtPos	public	TCustomListBox
ItemRect	public	TCustomListBox
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl

Method	Visibility	Documented in
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TDBListBox*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a *TDBListBox* object at runtime. For *TDBListBox* objects placed on forms or data modules at design time, *Create* is called automatically.

After calling the inherited constructor to allocate and initialize a generic list box object, *Create* creates and initializes the helper objects needed to link to a dataset.

See also *TCustomListBox.Create* method, *TFieldDataLink* object

Destroy

Destroy destroys an instance of *TDBListBox*.

destructor Destroy; **override**;

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TDBListBox* object is not already freed and only then calls *Destroy*.

Destroy frees the helper objects of the *TDBListBox* before calling the inherited destructor.

See also *Create* method, *Free* method

TDBLookupComboBox

Unit: DBCtrls | Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TDBLookupControl • TDBLookupComboBox

TDBLookupComboBox is a combo box that represents a set of field values in one dataset by a corresponding set of values from another dataset.

Description

Use *TDBLookupComboBox* to provide the user with a convenient drop-down list of lookup items for filling in fields that require data from another dataset.

If *TDBLookupComboBox* is linked to a lookup field component, it automatically reads the relationship between the field value and the lookup values in the lookup dataset from the field component. The relationship between field values and the corresponding values in the lookup dataset can also be explicitly set using the properties of the lookup combo box when the combo box is not linked to a lookup field component.

See also

TComboBox object, *TDBLookupListBox* object

Properties

The following table lists the properties of *TDBLookupComboBox*:

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl

D

Property	Visibility	Default	Documented in
DataField	published		TDBLookupControl
DataSource	published		TDBLookupControl
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl
DropDownAlign	published	daLeft	TDBLookupComboBox
DropDownRows	published	7	TDBLookupComboBox
DropDownWidth	published	0	TDBLookupComboBox
Enabled	published	True	TControl
Field	public, read-only		TDBLookupControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
KeyField	published		TDBLookupControl
KeyValue	public		TDBLookupControl
Left	published		TControl
ListField	published		TDBLookupControl
ListFieldIndex	published	0	TDBLookupControl
ListSource	published		TDBLookupControl
ListVisible	public, read-only		TDBLookupComboBox
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ReadOnly	published	False	TDBLookupControl
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Text	public, read-only		TDBLookupComboBox
Top	published		TControl
VCLComObject	public		TComponent

Property	Visibility	Default	Documented in
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

DropDownAlign

DropDownAlign specifies how the drop-down list is aligned relative to its edit box.

```
type TDropDownAlign = (daLeft, daRight, daCenter);
property DropDownAlign: TDropDownAlign;
```

Description Use *DropDownAlign* to specify the position of the drop-down list relative to the edit region of the lookup combo box. The drop down list can be aligned to the left or right, or centered with the edit box.

See also *DropDownWidth* property, *DropDownRows* property, *OnDropDown* event

DropDownRows

DropDownRows is the number of items displayed in the drop-down list of the *TDBLookupComboBox*.

```
property DropDownRows: Integer;
```

Description Use *DropDownRows* to specify how many rows appear in the drop down list of the combo box. If there are more lookup items than *DropDownRows*, the lookup list box displays a scroll bar.

See also *DropDownWidth* property, *DropDownAlign* property, *OnDropDown* event

DropDownWidth

DropDownWidth is the width, in pixels, of the drop-down list.

```
property DropDownWidth: Integer;
```

Description Use *DropDownWidth* to customize the width of the drop-down list. If *DropDownWidth* is 0 (the default), the drop-down list is the same width as the combo box.

DropDownWidth is useful when displaying multiple lookup fields, and therefore, multiple columns in the database lookup combo box.

See also *DropDownRows* property, *DropDownAlign* property, *OnDropDown* event

ListVisible

ListVisible specifies whether the lookup list is open or "dropped-down".

property ListVisible: Boolean;

Description Read *ListVisible* to determine whether the list of lookup values is currently in the open (dropped-down) position. If *ListVisible* is *True*, the list is open; if *ListVisible* is *False*, the list is closed.

See also *OnDropDown* event

Text

Text is the lookup value that corresponds to the value of the field on the current record.

property Text: **string**;

Description Read *Text* to determine the currently selected lookup value. *Text* is the value that appears in the edit region of the lookup combo box. The user can change the value of *Text* by entering a new value if *ReadOnly* is *False* and the dataset is in edit mode.

See also *TDataSet.Edit* method, *ReadOnly* property

Events

The following table lists the events of *TDBLookupComboBox*:

Event	Visibility	Documented in
OnClick	published	TControl
OnCloseUp	published	TDBLookupComboBox
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnDropDown	published	TDBLookupComboBox
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

OnCloseUp

OnCloseUp occurs immediately after an opened or "dropped-down" list is closed.

property OnCloseUp: TNotifyEvent;

Description Write an *OnCloseUp* event handler to respond when the combo box list is closed. When the list is closed, the value that corresponds to the selected lookup value is assigned to the field. The list can be closed by the user or by calling the *CloseUp* method

See also *CloseUp* method, *OnDropDown* method

OnDropDown

OnDropDown occurs immediately before the lookup list is opened.

property OnDropDown: TNotifyEvent;

Description Write an *OnDropDown* event handler to take specific action before the lookup list is displayed to the user. The list can be opened by the user or by calling the *DropDown* method.

See also *DropDown* method, *OnCloseUp* event

Methods

The following table lists the methods of *TDBLookupComboBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
CloseUp	public	TDBLookupComboBox
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDBLookupComboBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDBLookupControl
DestroyComponents	public	TComponent
Destroying	public	TComponent

Method	Visibility	Documented in
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
DropDown	public	TDBLookupComboBox
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl

Method	Visibility	Documented in
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

CloseUp

CloseUp closes an opened or "dropped-down" list.

```
procedure CloseUp(Accept: Boolean);
```

Description Call *CloseUp* to programmatically close the list of the lookup combo box. The *Accept* parameter determines whether to modify the field value with the value that corresponds to the selected value in the lookup list.

See also *DataField* property, *OnCloseUp* event

Create

Create creates an instance of *TDBLookupComboBox*.

```
constructor Create(AOwner: TComponent); override;
```

Description Call *Create* to instantiate a *TDBLookupComboBox* object at runtime. For *TDBLookupComboBox* objects placed on forms or data modules at design time, *Create* is called automatically.

After calling the inherited constructor to allocate and initialize a generic list box object, *Create* creates and initializes various helper objects.

See also *TDBLookupControl.Create* method, *TFieldDataLink* object

DropDown

DropDown opens or "drops down" the lookup list so that the user can choose a lookup value.

```
procedure DropDown;
```

Description Call *DropDown* to programmatically open the lookup list. Before the list is displayed, an *OnDropDown* event is generated.

See also *OnDropDown* event

TDBLookupControl

Unit: DBCtrls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TDBLookupControl

TDBLookupControl is an abstract base class for controls that provide the user with a convenient list of lookup items for filling in fields that require data from another dataset.

Description Use *TDBLookupControl* as a base class when defining controls that permit the user to set a field value by selecting a corresponding value from another dataset. The *TDBLookupComboBox* and *TDBLookupListBox* controls are both descended from the *TDBLookupControl* and provide the user with automatic lookup capabilities that are implemented in the *TDBLookupControl* object.

Do not create instances of *TDBLookupControl*. Instead, instantiate a descendant of *TDBLookupControl* to allow the user to set field values from a set of lookup values.

See also *TDBLookupComboBox* object, *TDBLookupListBox* object

Properties

The following table lists the properties of *TDBLookupControl:*

Property	Visibility	Default	Documented in
Align	public	alNone	TControl
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl

Property	Visibility	Default	Documented in
Cursor	published	crDefault	TControl
DataField	protected		TDBLookupControl
DataSource	protected		TDBLookupControl
DesignInfo	public		TComponent
Enabled	public	True	TControl
Field	public, read-only		TDBLookupControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
KeyField	protected		TDBLookupControl
KeyValue	protected		TDBLookupControl
Left	published		TControl
ListField	protected		TDBLookupControl
ListFieldIndex	protected	0	TDBLookupControl
ListSource	protected		TDBLookupControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	protected	False	TControl
ParentWindow	public		TWinControl
ReadOnly	protected	False	TDBLookupControl
ShowHint	public		TControl
Showing	public, read-only		TWinControl
TabOrder	public	–1	TWinControl
TabStop	protected	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	public	True	TControl
Width	published		TControl
WindowProc	public		TControl

DataField

DataField identifies the field whose value is represented by the lookup control.

property DataField: **string**;

Description Use *DataField* to bind the lookup control to a field in the dataset specified by the *DataSource* property. The *DataField* is the field whose value can be set by the lookup control, not the field which supplies the lookup values that are displayed by the lookup control.

D

If *DataField* specifies a lookup field component, there is no need to set the *KeyField* or *ListField* properties, as the field component contains all the information about the relationship between the data field value and the lookup value.

See also *DataSource* property, *KeyField* property, *ListField* property

DataSource

DataSource links the lookup control to the dataset that contains the *DataField*.

```
property DataSource: TDataSource;
```

Description Use *DataSource* to specify the data source component that identifies the dataset the lookup control represents. The lookup control represents a field in one table by displaying the value of a corresponding field in another table. *DataSource* links to the dataset for the field the control represents, as opposed to the *ListSource*, which links the lookup control to the lookup table that holds the field that is actually displayed. If the lookup control is used to edit data, the field that actually gets changed is the one in the *DataSource*.

See also *DataField* property, *ListSource* property

Field

Field is the *TField* object the lookup control represents.

```
property Field: TField;
```

Description Use the *Field* object to directly access the field component for the field the lookup control represents. If *Field* is a lookup field, the properties of the field component describe the relationship between the field and the lookup dataset. To allow a lookup control to represent a set of lookup values that do not come from another dataset, use the *DataField* property to bind to a lookup field and use the *LookupList* property of the *TField* object to specify a list of lookup values.

See also *DataField* property, *DataSource* property, *TField* object, *TField.LookupList* property

KeyField

KeyField identifies the field in the *ListSource* dataset that must match the value of the *DataField* field.

```
property KeyField: string;
```

Description Use *KeyField* to link the *ListSource* of the lookup control to the *DataSource*. Although the name of the field specified as the *KeyField* does not have to be the same as the name of the field specified as the *DataField*, the two fields must have the same values.

After specifying the *ListField*, choose which field the lookup control will actually display with the *ListField* property.

If the *DataField* field is a lookup field, don't specify anything for *KeyField* or *ListField*; lookup controls automatically use the data field's *LookupKeyFields* property for *KeyField*.

See also *DataField* property, *DataSource* property, *KeyValue* property, *ListField* property, *ListSource* property, *TField.LookupKeyFields* property

KeyValue

The *KeyValue* property is the common value of the *KeyField* field and the *DataField* field.

```
property KeyValue: Variant;
```

Description Use *KeyField* to determine the value represented by the lookup control (not the value displayed by the lookup control). When *KeyValue* is set, the lookup control attempts to find a record from the *ListSource*'s dataset where the value of *KeyField* matches *KeyValue*. If such a match is found, the lookup control displays the value of *ListField* on that record.

See also *DataField* property, *KeyField* property, *ListField* property, *ListSource* property, *TField.LookupKeyFields* property

ListField

ListField identifies the field or fields whose values are displayed in the lookup control.

```
property ListField: string;
```

Description The lookup control represents a field in one table by displaying the value of one or more corresponding fields in another table. *ListField* is the name of the field or fields in the lookup table that are actually displayed, as opposed to the *DataField*, which is the field in the *DataSource* that the lookup control actually represents, or the *KeyField*, which is the field in the lookup table with the same value as the *DataField*.

To fully specify the list fields, both a dataset for the lookup table and the fields within that dataset must be defined. The *ListSource* property of the lookup control specifies the dataset for the lookup table.

ListField can represent more than one field. Multiple field names should be separated by semicolons.

Before specifying *ListField*, specify the link between the two datasets using the *KeyField* property. If *ListField* is not set, lookup controls display *KeyField* field values by default. If the *DataField* field is a lookup field, don't specify anything for *ListField*; the data controls automatically use the lookup field's *LookupResultField* property as *ListField*.

See also *DataField* property, *DataSource* property, *KeyField* property, *KeyValue* property, *ListFieldIndex* property, *ListSource* property

ListFieldIndex

ListFieldIndex specifies which field from the *ListField* property is used for incremental searching.

```
property ListFieldIndex: Integer;
```

Description When the *ListField* property specifies more than one field, use *ListFieldIndex* to specify which of those fields is the one to use for incremental searches. For the *TDBLookupComboBox* object, *ListFieldIndex* also determines which field appears in the edit region of the combo box.

ListFieldIndex allows the most important value to appear in a position other than the first when all the values of the *ListField* fields are displayed. The value of *ListFieldIndex* must be less than the number of fields specified by the *ListField* property.

See also *Listfield* property

ListSource

ListSource identifies a data source for the data displayed in the lookup control.

```
property ListSource: TDataSource;
```

Description Set *LookupSource* to the data source that contains the *KeyField* and *ListField* fields. If the *DataField* field is a lookup field, don't specify anything for *ListSource*; data controls automatically use the lookup field's *LookupDataSet* property to create a data source.

See also *DataField* property, *DataSource* property, *KeyField* property, *ListField* property, *ListFieldIndex* property, *TField.LookupDataSet* property

ReadOnly

ReadOnly determines if the user can use the lookup control to set the value of the *DataField* field.

```
property ReadOnly: Boolean;
```

Description Use *ReadOnly* to specify whether the lookup control is used for display purposes only. If *ReadOnly* is *True*, the lookup control can only be used to display the lookup values that correspond to the value of the *DataField* field on the current record. If *ReadOnly* is *False*, the user can use the lookup control to change the field's value.

See also *DataField* property, *TDataSet.Edit* method

Methods

The following table lists the methods of *TDBLookupControl*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDBLookupControl
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDBLookupControl
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent

Method	Visibility	Documented in
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
Notification	protected	TDBLookupControl
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TDBLookupControl*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a *TDBLookupControl* object at runtime. For *TDBLookupControl* objects placed on forms or data modules at design time, *Create* is called automatically.

After calling the inherited constructor to allocate and initialize a generic custom control, *Create* creates and initializes the helper objects needed to link to the datasets.

See also *TCustomListBox.Create* method, *TDataSourceLink* object, *TListSourceLink* object

Destroy

Destroy destroys an instance of *TDBLookupControl*.

```
destructor Destroy; override;
```

Description Do not call *Destroy* directly in an application. Instead, call *Free*. *Free* verifies that the *TDBLookupControl* object is not already freed and only then calls *Destroy*.

Destroy frees the helper objects of the *TDBLookupControl* before calling the inherited destructor.

See also *Create* method, *Free* method

Notification

Notification overrides the inherited method to respond to *opRemove* notifications on the *DataSource* and *ListSource*.

```
type TOperation = (opInsert, opRemove);
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Description After calling the inherited method, *Notification* checks whether the notification indicates the removal of *DataSource* or *ListSource*. If so, *Notification* sets the property to **nil**.

Derived classes can override *Notification* method to handle additional notifications, but the inherited method should be always be called.

See also *DataSource* property, *ListSource* property

TDBLookupListBox

Unit: DBCtrls — Palette: Data Controls

TObject • TPersistent • TComponent • TControl • TWinControl • TCustomControl • TDBLookupControl • TDBLookupListBox

TDBLookupListBox provides a list of lookup items for filling in fields that require data from another dataset.

Description Use *TDBLookupListBox* to provide users with a convenient list of lookup items to set a field value using the values of a field in another dataset. Lookup list boxes usually display values that are a represent a more meaningful description of the actual field value.

If *TDBLookupListBox* is linked to a lookup field component, it automatically reads the relationship between the field value and the lookup values in the lookup dataset from the field component. The relationship between field values and the corresponding values in the lookup dataset can also be explicitly set using the properties of the lookup list box when the list box is not linked to a lookup field component.

See also *TDBLookupComboBox* object, *TDBListBox* object

Properties

The following table lists the properties of *TDBLookupListBox*:

Property	Visibility	Default	Documented in
Align	published	alNone	TControl
BorderStyle	published	bsSingle	TDBLookupListBox
BoundsRect	public		TControl
Brush	public, read-only		TWinControl
ClientHeight	public		TControl
ClientOrigin	public, read-only		TWinControl
ClientRect	public, read-only		TWinControl
ClientWidth	public		TControl
Color	published	clWindow	TControl
ComObject	public, read-only		TComponent
ComponentCount	public, read-only		TComponent
ComponentIndex	public		TComponent
Components	public, read-only		TComponent
ComponentState	public, read-only		TComponent
ComponentStyle	public, read-only		TComponent
ControlCount	public, read-only		TWinControl
Controls	public, read-only		TWinControl
ControlState	public		TControl
ControlStyle	public		TControl
Ctl3D	published		TWinControl
Cursor	published	crDefault	TControl
DataField	published		TDBLookupControl
DataSource	published		TDBLookupControl
DesignInfo	public		TComponent
DragCursor	published	crDrag	TControl
DragMode	published	dmManual	TControl

Property	Visibility	Default	Documented in
Enabled	published	True	TControl
Field	public, read-only		TDBLookupControl
Font	published		TControl
Handle	public, read-only		TWinControl
Height	published		TControl
HelpContext	published	0	TWinControl
Hint	published		TControl
ImeMode	published	imDontCare	TWinControl
ImeName	published		TWinControl
KeyField	published		TDBLookupControl
KeyValue	public		TDBLookupControl
Left	published		TControl
ListField	published		TDBLookupControl
ListFieldIndex	published	0	TDBLookupControl
ListSource	published		TDBLookupControl
Name	published		TControl
Owner	public, read-only		TComponent
Parent	public		TControl
ParentColor	published	False	TControl
ParentCtl3D	published	True	TWinControl
ParentFont	published	True	TControl
ParentShowHint	published	True	TControl
ParentWindow	public		TWinControl
PopupMenu	published		TControl
ReadOnly	published	False	TDBLookupControl
RowCount	published		TDBLookupListBox
SelectedItem	public, read-only		TDBLookupListBox
ShowHint	published		TControl
Showing	public, read-only		TWinControl
TabOrder	published	–1	TWinControl
TabStop	published	True	TWinControl
Tag	published	0	TComponent
Top	published		TControl
VCLComObject	public		TComponent
Visible	published	True	TControl
Width	published		TControl
WindowProc	public		TControl

BorderStyle

BorderStyle determines whether the lookup list box has a border.

property BorderStyle: TBorderStyle;

Description Set *BorderStyle* to specify whether a border should be drawn around the lookup list box. These are the possible values:

Value	Meaning
bsNone	No visible border
bsSingle	Single-line border

See also *Ctl3D* property

ReadOnly

ReadOnly determines whether the user can change the field value using the lookup list box.

```
property ReadOnly: Boolean;
```

Description Set *ReadOnly* to specify whether the lookup list box should be used for display purposes only. If *ReadOnly* is *True*, the list box can only be used to display the lookup value that corresponds to the field's value on the current record. If *ReadOnly* is *False*, the user can use the list box to change the field's value.

See also *TDataset.Edit* method

RowCount

RowCount specifies how many rows are visible in the list box.

```
property RowCount: Integer;
```

Description Set *RowCount* to the number of rows displayed in the lookup list box. The list box is resized to match the specified number of rows. Resizing the list box (such as by changing its *Height* property) automatically changes the value of *RowCount* to match the new height.

See also *Height* property

SelectedItem

SelectedItem contains the value displayed in the list box that is currently selected.

```
property SelectedItem: string;
```

Description Read *SelectedItem* to determine the value the user selected using the lookup list box. If no item is selected, *SelectedItem* is an empty string.

See also *KeyField* property

Events

The following table lists the events of *TDBLookupListBox*:

Event	Visibility	Documented in
OnClick	published	TControl
OnDblClick	published	TControl
OnDragDrop	published	TControl
OnDragOver	published	TControl
OnEndDrag	published	TControl
OnEnter	published	TWinControl
OnExit	published	TWinControl
OnKeyDown	published	TWinControl
OnKeyPress	published	TWinControl
OnKeyUp	published	TWinControl
OnMouseDown	published	TControl
OnMouseMove	published	TControl
OnMouseUp	published	TControl
OnStartDrag	published	TControl

Methods

The following table lists the methods of *TDBLookupListBox*:

Method	Visibility	Documented in
Assign	public	TPersistent
BeginDrag	public	TControl
BringToFront	public	TControl
Broadcast	public	TWinControl
CanFocus	public	TWinControl
ClassInfo	public	TObject
ClassName	public	TObject
ClassNameIs	public	TObject
ClassParent	public	TObject
ClassType	public	TObject
CleanupInstance	public	TObject
ClientToScreen	public	TControl
ContainsControl	public	TWinControl
ControlAtPos	public	TWinControl
Create	public	TDBLookupListBox
CreateParented	public	TWinControl
DefaultHandler	public	TObject
Destroy	public	TDBLookupControl

Method	Visibility	Documented in
DestroyComponents	public	TComponent
Destroying	public	TComponent
DisableAlign	public	TWinControl
Dispatch	public	TObject
DragDrop	public	TControl
Dragging	public	TControl
EnableAlign	public	TWinControl
EndDrag	public	TControl
FieldAddress	public	TObject
FindComponent	public	TComponent
Focused	public	TWinControl
Free	public	TObject
FreeInstance	public	TObject
FreeNotification	public	TComponent
FreeOnRelease	public	TComponent
GetInterface	public	TObject
GetInterfaceEntry	public	TObject
GetInterfaceTable	public	TObject
GetNamePath	public	TPersistent
GetParentComponent	public	TComponent
GetTabOrderList	public	TWinControl
GetTextBuf	public	TControl
GetTextLen	public	TControl
HandleAllocated	public	TWinControl
HandleNeeded	public	TWinControl
HasParent	public	TComponent
Hide	public	TControl
InheritsFrom	public	TObject
InitInstance	public	TObject
InsertComponent	public	TComponent
InsertControl	public	TWinControl
InstanceSize	public	TObject
Invalidate	public	TWinControl
MethodAddress	public	TObject
MethodName	public	TObject
NewInstance	public	TObject
PaintTo	public	TWinControl
Perform	public	TControl
Realign	public	TWinControl
Refresh	public	TControl
RemoveComponent	public	TComponent
RemoveControl	public	TWinControl

Method	Visibility	Documented in
Repaint	public	TWinControl
SafeCallException	public	TComponent
ScaleBy	public	TWinControl
ScreenToClient	public	TControl
ScrollBy	public	TWinControl
SendToBack	public	TControl
SetBounds	public	TWinControl
SetFocus	public	TWinControl
SetTextBuf	public	TControl
Show	public	TControl
Update	public	TWinControl
UpdateControlState	public	TWinControl

Create

Create creates an instance of *TDBLookupListBox*.

constructor Create(AOwner: TComponent); **override**;

Description Call *Create* to instantiate a *TDBLookupListBox* object at runtime. For *TDBLookupListBox* objects placed on forms or data modules at design time, *Create* is called automatically.

After calling the inherited constructor to allocate and initialize a generic lookup control, *Create* creates and initializes the helper objects needed to link to a dataset.

See also *TDBLookupControl.Create* method, *TFieldDataLink* object

D

Borland
www.borland.com

Copyright © 1997 Borland International, Inc. All rights reserved. All Borland product names are trademarks of Borland International, Inc. Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Internet: http://www.borland.com CompuServe: GO BORLAND. Offices in: Australia, Canada, France, Germany, Hong Kong, Japan, Latin America, Mexico, The Netherlands, Taiwan, and United Kingdom • Part # HDA1330WW21773 • BOR 9962